The suitability of CRSX for transformations on Haskell

Koen Rutten
Utrecht University

September 20, 2012

Bachelor's thesis (15 ECTS)

Supervisors: Prof. Dr. Vincent van Oostrom
Utrecht University

Kristoffer H. Rose, Ph.D.
IBM Thomas J. Watson Research Center

Abstract

When compiling a program, the code is transformed several times, optimizing the code or translating to another language. Some of these transformations are, or almost are, CRSs (Combinatory Reduction Systems). CRSs are abstract systems with a lot of theoretical background: fitting a transformation in CRS can for example help prove correctness of the transformation. In recognition of the benefit CRSs can have to the compilation process, CRSX was developed: a practical implementation of CRS with extensions. This paper investigates the suitability of CRS and CRSX for transformations on programs written in Haskell.

Contents

1 Introduction

This thesis details my research on the suitability of CRSX for transformations on Haskell. I will start with a very brief introduction of both Haskell and CRSX before I present my approach on this subject.

1.1 Haskell

Haskell is a purely functional programming language, meaning that programs are defined mainly in terms of functions without side effects. Haskell is a very rich language, but most of its constructs can be translated to simple λ-calculus extended with a recursive Let construction. Haskell and this translation are presented more extensively in Section 3.

λ-calculus, or lambda-calculus, is a simple formalism to express pure functions. For example, here is a definition of the square function, applied on the number 3:

```
(λ x . x * x) 3
```

The λ signals the start of a function definition, followed by the parameters, a dot, and the body of the function. The entire term is just a single expression, and would have value `9` if evaluated. Actually, this is not a "pure" λ expression, because then there would be only functions. In this example, numbers are added to the calculus. λ-calculus and the needed extensions are described in Section 2.

1.2 CRSX

In an expression like `λ x . x * x`, the `x`'s in the body refer to the parameter called `x`, introduced at the start of the λ-term. Such a construction is called "variable binding", here `x` is the *bound variable*, and λ is the *binder*. (In λ-calculus, λ is the only binder.)

Combinatory reduction systems (CRSs) are rewriting systems designed to rewrite terms with bound variables, such as λ-calculus. Such a system consists of a number of patterns and replacements. In our case, if a term is put into the system, it will be searched for any occurrences of the patterns, and the matched subterm will be replaced by the corresponding replacement. This is repeated until no further replacements can be made, and the resulting term is the output. An exact explanation of this process is given in Section 4.3.

CRSX (Combinatory Reduction Systems with Extensions) is an implementation of a CRS engine with certain extensions. It's designed to be used as (a part of) a compiler. More details are in Section 4.4.

1.3 Using CRSX for transformations on Haskell code

Compilation generally consists of several steps: translation to an intermediate language, optimizing transformations on the intermediate language, and finally generating machine code from this. All of these steps can be seen as rewrite steps, but that doesn't mean they can be expressed using CRSX. For Haskell, λ-calculus can be used as intermediate language. In my research, I focus on optimizing transformations on λ-calculus, and examine if they can be done with CRSX.

1.4 Artificial Intelligence

This thesis was written as part of my bachelor's degree in Artifical Intelligence. One way to define the study of Artifical Intelligence is that it investigates the structure in processes of (human) intelligence. Researching the suitability of CRSX for transformations on Haskell is basicly a search for structure: I try to determine if structure in CRS is also part of Haskell transformations.

2 λ-calculus

Functional programming is done by specifying functions. These functions can take as parameters not only ordinary values, but also functions. The result of a function, too, can be a function. As we will see, λ-calculus is a simple system to express such functions. It was introduced by Alonzo Church in 1936, and has become a very widely used system. We will give an introduction to λ-calculus that is sufficient for our research. Numerous others can be found, there is one in [Terese, 2003].

In pure λ-calculus, there exist only functions: parameters, and the results of functions, are all functions. It is possible to represent all kinds of objects as functions, including numbers and operations on numbers; one such way is Church encoding. This means that pure λ-calculus is powerful enough to use as a model for computation, but it's not very intuitive. To make for a more practical language, we will add (integer) numbers. To calculate with these newly added values, we will also add addition, subtraction and multiplication, and a zero-test function.

2.1 Writing programs in λ-calculus

Now that we have the ingredients for our λ-calculus, let's see what it looks like. To start with, here is an expression representing the square function:

```
λ x . x * x
```

The λ indicates that the expression is a function. It is followed by the parameter name, a dot, and then another expression: the body of the function. An expression that is a function, defined with a λ, is called an *abstraction.* This does not include the built-in functions like multiplication.

To evaluate a function, it needs a value for the parameter. This value is supplied by an *application.* There is no special symbol for application, we just write the value for the parameter after the function. For example, the following application would evaluate to 9:

```
(λ x . x * x) 3
```

The parentheses are added to make it clear where the function body ends.

2.2 Evaluation

Evaluation is what gives λ-calculus meaning. Evaluation of a term is done by repeatedly reducing subterms (including the whole term) according to one of the allowed reduction steps. For pure λ-calculus, only one kind of reduction is needed, called β-reduction: If a term is an application, and the left subterm is an abstraction, we can replace it by the body of the function, in which we replace all occurrences of the parameter with the right subterm.

For example, β-reduction can be used on the term given earlier:

$$\texttt{(}\lambda\texttt{ x . x * x) 3} \rightarrow_\beta \texttt{3 * 3}$$

This also shows that to extend λ-calculus, the added functions need their own reduction steps. These are called δ-reductions. For arithmetic functions this is straightforward: They can be reduced when their arguments are numbers:

$$\texttt{(}\lambda\texttt{ x . x * x) 3} \rightarrow_\beta \texttt{3 * 3} \rightarrow_\delta \texttt{9}$$

In this example, only one reduction is possible at every step, until no more reductions are possible and the evaluation is done. However, in general there may be multiple reductions possible on an expression. At this point, nothing is specified about which reduction to choose, and this isn't even very interesting: Just specifying the possible reductions gives λ-calculus a reasonably exact meaning. For example, if a term can be reduced to a single number, it can not be reduced to another number using a different series of reductions. In constructing an actual programming language however, the order is important, as we will see in Section 3.1.

2.3 Notational shortcuts

In λ-calculus, a function can have only one parameter. To represent a function of multiple parameters, we can use a "trick" as follows: Say we want to build a function of two parameters, x and y, that calculates `(x + y) * (x - y)`. Instead we construct a function, taking x as parameter, that returns another function, with parameter y and returning the intended result. Here's an example of how this looks in λ-calculus:

```
λ x . (λ y . (x + y) * (x - y))
```

Because this construction is used a lot, a shortcut notation is generally used. The above example can be written shorter as:

```
λ x y . (x + y) * (x - y)
```

The shortcut notation can be used whenever the body of an abstraction is another abstraction. The notation for nested applications is simplified too, by defining application to associate to the left. This means that if the left part of an application is another application, it doesn't need parentheses, because an expression like `a b c d ...` is taken to mean `(((a b) c) d) ...`.

The following reduction sequence uses both shortcut notations. It also uses a double headed arrow to hide some intermediate reduction steps. The meaning of this notation will be made more precise in Section 4.1.

$$\begin{aligned}
&\texttt{(}\lambda\texttt{ x y . (x + y) * (x - y)) 5 3}\\
\rightarrow_\beta\ &\texttt{(}\lambda\texttt{ y . (5 + y) * (5 - y)) 3}\\
\rightarrow_\beta\ &\texttt{(5 + 3) * (5 - 3)}\\
\twoheadrightarrow_\delta\ &\texttt{16}
\end{aligned}$$

The expression on the first line can be thought of as a function with two parameters, which is supplied two values. Semanticly, this is a useful interpretation, because through using β-reduction twice, the arguments can be given the corresponding values as expected. But it is important to remember the expanded meaning of the shortcut notation: Since β-reduction can only be used when an application and an abstraction are directly nested, the only possible reduction on the first line is the shown one, substituting the parameter `x`. Then after this reduction, `y` can be substituted.

It may make sense to substitute the parameter `y` before or even without substituting `x`: the result would still evaluate to the same value. While such a reduction is not possible using β-reduction, it is indeed a sensible reduction step, and is called *generalized β-reduction.* This concept will be investigated further in Section 7.

Introducing shortcut notation has a disadvantage: It allows for multiple ways to write a single expression. To mitigate this, in this thesis, the shortcuts will be used wherever possible. This may not always be the obvious choice, one may for example define a function for function composition as:

```
λ f g . (λ x . f (g x))
```

But full use of shortcut notation demands this to be written as:

```
λ f g x . f (g x)
```

2.4 Zero-test function

The zero-test function is called `Zero` and can be used to test if a number is `0`. The reduction rules are:

$$\begin{array}{lll} \texttt{Zero 0} \rightarrow_\delta \lambda\ \texttt{x y . x} & & \\ \texttt{Zero \#} \rightarrow_\delta \lambda\ \texttt{x y . y} & & \textit{when } \texttt{\#} \textit{ is any integer other than } \texttt{0} \end{array}$$

Although `Zero` is a function of one parameter, it's easier to think of it as having three parameters: First an integer, then the value for the case the integer is zero, and then the value for the other case. This is shown by the following reduction sequences, where $\#_0$ and $\#_1$ can be any λ-calculus expression:

$$\begin{array}{ll} & \texttt{Zero 0 } \#_0\ \#_1 \\ \rightarrow_\delta & (\lambda\ \texttt{x y . x})\ \#_0\ \#_1 \\ \rightarrow_\beta & (\lambda\ \texttt{y . } \#_0)\ \#_1 \\ \rightarrow_\beta & \#_0 \end{array}$$

$$\begin{array}{ll} & \texttt{Zero 1 } \#_0\ \#_1 \\ \rightarrow_\delta & (\lambda\ \texttt{x y . y})\ \#_0\ \#_1 \\ \rightarrow_\beta & (\lambda\ \texttt{y . y})\ \#_1 \\ \rightarrow_\beta & \#_1 \end{array}$$

2.5 α-conversion

The square function was previously expressed as:

```
λ x . x * x
```

But it could be equally well expressed as:

```
λ y . y * y
```

In general, renaming the parameter of an abstraction, and the variables referring to it, does not change the meaning of an expression. This process is called α-conversion. There is one important condition though: if the new name is already used in the body of the function by another abstraction, renaming is not allowed, because in that case the meaning may change. For example, in the function used earlier:

```
λ x y . (x + y) * (x - y)
```

parameter `x` may not be renamed to `y`, nor `y` to `x`. Determining when an α-conversion using an existing name is safe is not very complicated, but the easiest way is always choosing a name that is not yet used in the entire expression, a so called *fresh* variable.

An expression is allowed to use two parameters with the same name. An abstraction introduces a parameter by giving it a name, and only in the body of the abstraction this name can be used to refer to the parameter. So if two abstractions with the same parameter name are not nested, there is no source for confusion. In the case one abstraction is inside the body of the other abstraction, the inner abstraction wins: In the body of the inner abstraction, occurrences of the name refer to the parameter of the inner abstraction, and equally named parameters of outer abstractions cannot be referenced.

Renaming variables is sometimes necessary during evaluating, consider for example the following reduction sequence:

```
    (λ f . f f) (λ a b . a b)
→β  (λ a b . a b) (λ a b . a b)
→β  (λ b . (λ a b . a b) b)
→β  λ b b . b b    [WRONG]
```

On the first line, all parameters have different names. On the second line, the names `a` and `b` are used twice, but there is no problem, because the abstractions using the same name are not nested in each other. On the third line, there are two nested abstractions using the name `b`, but there is still no problem, although it needs to be interpreted with care. However, the last reduction is problematic: The last `b` in the expression replaces the `a`, but inside the body of the `a`-abstraction, the name `b` has another meaning.

This problem is called *name capturing.* The solution is to use α-conversion before a problematic β-reduction. Using a fresh variable is easiest, one can be created by adding an index to the original name, or if it already has an index, increasing it. This gives the following, valid, evaluation of the above example:

$$\begin{array}{ll}
 & (\lambda\ \mathtt{f}\ .\ \mathtt{f\ f})\ (\lambda\ \mathtt{a\ b}\ .\ \mathtt{a\ b}) \\
\to_\beta & (\lambda\ \mathtt{a\ b}\ .\ \mathtt{a\ b})\ (\lambda\ \mathtt{a\ b}\ .\ \mathtt{a\ b}) \\
\to_\beta & (\lambda\ \mathtt{b}\ .\ (\lambda\ \mathtt{a\ b}\ .\ \mathtt{a\ b})\ \mathtt{b}) \\
\to_\alpha & (\lambda\ \mathtt{b}\ .\ (\lambda\ \mathtt{a}\ \mathtt{b}_1\ .\ \mathtt{a}\ \mathtt{b}_1)\ \mathtt{b}) \\
\to_\beta & \lambda\ \mathtt{b}\ \mathtt{b}_1\ .\ \mathtt{b}\ \mathtt{b}_1
\end{array}$$

The fresh variable is simply called $\mathtt{b}_1$, and should the need arise for more fresh variables based on `b` or $\mathtt{b}_1$, they will be called $\mathtt{b}_2$, $\mathtt{b}_3$ etc. Whenever a β-reduction needs a renaming, α-conversion will be implicitly used as part of the reduction. We assume an infinite supply of fresh variables for this.

A more elegant solution to the problem of name capturing is to use the set of expressions modulo α-conversion. In this view, renaming just offers the possibility of writing down the *same* abstraction in different ways, but the abstractions themself are nameless; renaming and name capture do no exists. This view is actually the most accurate, and expressions can be represented in a computer this way using object references. But because we need to name things to write about them, and to let the computer output them in a readable format, the hassle of α-conversion can not be entirely avoided. It is useful however, to think of bound variables in this way.

2.6 Recursion

A simple extension to λ-calculus is the non-recursive `Let` construction. It is an expression taking the form `Let v =` $\#_d$ `In` $\#_e$ where `v` is a variable and $\#_d$ and $\#_e$ are λ-expressions. For example, it can be used to define a function to calculate `a * a - b * b` as:

```
λ a b . Let square = λ x . x * x In square a - square b
```

The meaning of a `Let` construction is straightforward: it's value is the expression $\#_e$, where inside $\#_e$, `v` refers to $\#_d$. This is the same as an application and an abstraction, and the above example could be rewritten as:

```
λ a b . (λ square . square a - square b) (λ x . x * x)
```

In general, the `Let` construction can be replaced by `(λ v .` $\#_e$`)` $\#_d$.

The *recursive* `Let` construction, also called *LetRec*, is similar to the non-recursive one, with one important difference: not only in the expression $\#_e$, but also in $\#_d$, `v` refers to $\#_d$. This allows for recursive function definitions.

Here, and later, we will use the factorial function as an example of a recursive function. It is defnined for non-negative integers: The factorial of `n` equals the product of the integers from `1` upto and including `n`. A recursive definition can be given by:

$$\text{fac}(n) = \begin{cases} 1 & \text{when } n = 0 \\ n \times \text{fac}(n-1) & \text{when } n > 0 \end{cases}$$

This can be expressed in λ-calculus, using a recursive `Let`, as:

```
Let fac = λ n . Zero n 1 (n * (fac (n - 1)))
In  fac 5
```

Which should evaluate to something equal to `5 * 4 * 3 * 2 * 1 = 120`. The required reduction rule is:

`Let v =` $\#_d$ `In` $\#_e \rightarrow_L \#_e$`[v := (Let v =` $\#_d$ `In` $\#_d$`)]`

Here $\#_e$`[v := r]` stands for the expression $\#_e$ where all bound occurrences of `v` are replaced by `r`.

To see how the factorial example can be reduced, we'll shorten an expression that occurs often during the reduction: Let F stand for the expression:

```
Let fac = λ n . Zero n 1 (n * (fac (n - 1)))
In        λ n . Zero n 1 (n * (fac (n - 1)))
```

Note that F can be reduced to `λ n . Zero n 1 (n * (`F` (n - 1)))`. The factorial example can be reduced as follows:

$$\begin{array}{ll}
 & \texttt{Let fac = }\lambda\texttt{ n . Zero n 1 (n * (fac (n - 1))) In fac 5} \\
\to_L & F\ \texttt{5} \\
\to_L & \texttt{(}\lambda\texttt{ n . Zero n 1 (n * (}F\texttt{ (n - 1)))) 5} \\
\to_\beta & \texttt{Zero 5 1 (5 * (}F\texttt{ (5 - 1)))} \\
\twoheadrightarrow_{\beta\delta} & \texttt{5 * (}F\texttt{ 4)} \\
\to_L & \texttt{5 * ((}\lambda\texttt{ n . Zero n 1 (n * (}F\texttt{ (n - 1)))) 4)} \\
\to_\beta & \texttt{5 * (Zero 4 1 (4 * (}F\texttt{ (4 - 1))))} \\
\twoheadrightarrow_{\beta\delta} & \texttt{5 * (4 * (}F\texttt{ 3))} \\
\twoheadrightarrow_{\beta\delta L} & \texttt{5 * (4 * (3 * (}F\texttt{ 2)))} \\
\twoheadrightarrow_{\beta\delta L} & \texttt{5 * (4 * (3 * (2 * (}F\texttt{ 1))))} \\
\twoheadrightarrow_{\beta\delta L} & \texttt{5 * (4 * (3 * (2 * (1 * (}F\texttt{ 0)))))} \\
\twoheadrightarrow_{\beta\delta L} & \texttt{5 * (4 * (3 * (2 * (1 * 1))))} \\
\twoheadrightarrow_{\delta} & \texttt{120}
\end{array}$$

Even with the shorthand F, and hiding some in-between steps, it's quite a lengthy evaluation. There are actually 41 reductions made in this example. So let's see how we let a computer do the work for us.

3 Haskell

The Haskell programming language ([Peyton Jones et al., 2003]) is based on λ-calculus: A program is an expression, and running it is done by evaluating it. The expressions are like the λ-expressions from Section 2, but there are some differences. We will discuss these differences, but we start with an explanation of the evaluation process.

3.1 Evaluation

In λ-calculus, a function can be supplied only one argument. (Remember that `f x y` really means `(f x) y`, that is, `f x` is expected to reduce to a function, which in turn is supplied `y`.) There seems to be an exception for the functions `+` and `*`: the δ-reduction rules apply two arguments at once. Also they seem to enjoy a special notation: they are written between their arguments, in stead of before them, like the other functions in λ-calculus.

To simplify reasoning about the evaluation process, we will change this. We remove the functions `+` and `*`, and replace them with two new constants, `Plus` and `Times`. They are behave like regular functions, so instead of `2 * 3` we now write `Times 2 3`. The reduction rules for these new function use one argument at a time.

With this simplification, any λ-expression or subexpression is one of the following:

- An application
- An abstraction

- A variable
- A constant
- A `Let` contruction

This breakdown will be our lead for reasoning about λ-expressions. In general, β and δ-reduction only reduce (sub-)expressions that are applications, and the `Let` reduction rule reduces `Let` contructions. Other expressions can not be reduced.

Laziness

As we have seen, the reduction rules described for λ-calculus can be used to evaluate expressions, but we haven't discussed when to apply which rule: It's possible to have an expression that can be reduced in more than one way, using different rules, or even using a single rule applied to different parts of the expression.

The λ-calculus itself does not specify how the rules should be applied, only that evaluation is done when no more reductions can be made. As discussed in Section 2.2, a different choice of reductions won't give a different result. But the choice does determine how fast the result is reached, and even if it is reached at all: In some cases, an infinite series of reductions is possible.

An infinite reduction may be the only possibility, for example the expression `Let x = 1 + x In x` can only be rewritten as:

$$\begin{array}{ll} & \texttt{Let x = 1 + x In x} \\ \rightarrow_L & \texttt{1 + (Let x = 1 + x In x)} \\ \rightarrow_L & \texttt{1 + (1 + (Let x = 1 + x In x))} \\ \rightarrow_L & \dots \end{array}$$

But the factorial example from Section 2.6 can also be reduced infinitly with the right choice of reductions.

To be effective as a programming language, the reduction rules need to be more specific. The rules Haskell uses are *deterministic*: for any expression, at most one reduction is possible. As a consequence, a program can be evaluated in only one way. These rules are based on *lazy λ-calculus*, which is not λ-calculus, but a variation on it.

Lazy λ-calculus has the same terms as ordinary λ-calculus, only the reduction rules are different. In particular, the rules are stricter: some reductions are no longer possible, and no new reductions are allowed.

Lazy reduction rules

In λ-calculus, the reduction rules can be used on any subexpression; it doesn't matter where this subexpression is in the whole expression. This is not the case in lazy λ-calculus. Here, reduction is only allowed if the subexpression is eligible, as determined by the following rules:

- The entire expression is eligible.
- If an eligible subexpression is an application, the the left part is eligible.
- If an eligible subexpression is an application, and the left part is a constant, the right part is eligible.

Using these rules, there can be at most one reducable eligible subexpression. This can be inferred from the following two observations:

- An eligible subexpression has at most one eligible direct subexpression, and
- A reducable eligible subexpression has no eligible subexpressions that can be reducable:
  Because it is reducable, it fits the pattern of one the reduction rules, none of which make a subexpression eligible:
  - β: The subexpression is an application, but the left part is an abstraction.
  - δ: The subexpression is an application, and the left part is a constant, but so is the right side.
  - `Let`: The subexpression is a `Let`-construction.

After each reduction step, eligibility has to be reestablished for the updated expression. Note that because eligibility of a subexpression does not depend on the content of the subexpression itself, but rather of the rest of the expression. This means in particular that if an eligible subexpression is replaced in a reduction step, the replacement is also eligible. Also, because no subexpression of an abstraction can be eligible, an eligible subexpression can never be a variable.

The evaluation of lazy λ-calculus can also be expressed as a recursive imperative program. Running the program once is equivalent to using the reduction steps of lazy λ-calculus until no further reduction can be made.

Lazy evaluation algorithm

Input: A valid eligible λ-expression E.
Output: A λ-expression that is a either an abstraction or a constant.
The evaluation depends on the form of the expression E. The possibilities are:

- An application $E = AB$
  {* A is eligible because E is *}
  Evaluate A to A'.
  Proceed depending on the form of A'
  - A' is an abstraction:
    {* $A'B$ is eligible because it is at the same location as E was *}
    Use β-reduction to reduce $A'B$ to E'
    {* E' is eligible because it is at the same location as $A'B$ was *}
    Evaluate E'
  - A' is a constant:
    {* B is eligible because E was, en A' is a constant *}
    evaluate B to B'
    {* $A'B'$ is eligible because it is at the same location as E was *}
    δ-reduction on $A'B'$.
- An abstraction: done
- A variable: this can not happen because E is eligible.

- A constant: done
- A `Let` contruction:
  Reduce E to E' using the `Let` reduction rule.
  `{*` E' is eligible because it is at the same location as E was `*}`
  Evaluate E'

This program has comment between the statments, marked with `{* *}`. They describe the state of the program at the moment execution passes trough them. Using these comments it can be seen that the program only makes reductions on eligible subexpressions. Also, when the program stops, the output is an irreducable expression. To conclude that this program indeed evaluates expressions as specified by lazy λ-calculus, one more observation is needed: The inputs for which the program does not terminate coincide with the expressions that have an infinite reduction sequence. We omit the proof of this.

3.2 Differences with λ-calculus

Instead of studying Haskell itself, we will discuss how Haskell differs from λ-calculus. As we will see, the transformations on λ-expression we will investigate can be applied to Haskell code as well.

Types

The lazy evaluation algorithm as given in the previous section has one flaw: When it is stated that δ-reduction must be applied, this is not always possible. To use δ-reduction on $A'B'$, A' needs to be not just any constant, but one that represents a function. And B' needs to be an appropiate argument for this function. Two examples where the program would fail are `2 3` and `Plus (λ x . x)`. Letting the program crash on these inputs is not a bad solution: there is no sensible way to reduce these expressions. But Haskell has a more elegant solution to this problem: Type checking. In Haskell, every subexpression is assigned a *type*. For example, the type of any integer constant will be integer. Function get a type of the form $a \to b$, where a is the required type of the argument, and b is the type of the function result: `Plus` gets the type integer $\to$ (integer $\to$ integer). The programmer is allowed, but almost never required, to specify the types of any subexpression in his program. Using a *type inference algorithm*, Haskell will check if there is a way to assign types to the remaining subexpressions so that all functions are applied appropiate arguments. If it is successful, this will guarantee all δ-reductions will be possible, and evaluation can begin.

Haskell's type system means that some λ-expressions are not valid Haskell programs. This does not only include expressions where δ-reduction runs into trouble, but also certain expressions that could be properly evaluated. This is normally not really a problem for the programmer, but we have to be careful with this: If we interpet a Haskell program as a λ-expression, and then transform it into another λ-expression, this is not necessarily a valid Haskell program.

We will not run into such problems in the research in this thesis. A proof of this would require a procedure, for each of the transformations we will discuss, to coherently assign types to the transformed expression, based on the types of the input expression. Doing this would require an exposition of Haskell's type system, which falls outside the scope of this thesis, so we will not concern ourselfs with typing any further.

More constants

Haskell has a lot more to offer than just doing operations on integers. This includes:

- Additional data types, like floating point numbers and textual data.
- The definition of custom data types, and pattern matching on them.
- Functions to interact with the "world", such as the user, the network or the filesystem.
- Classes, to unify different datatypes that have a common subset of operations.

All of these can be accomodated in lazy λ-calculus using a combination of additional constants, and by viewing the used constructs as shortcut notations for regular λ-expressions. Luckily, the number of constants doesn't matter for the transformations we will discuss, the only importance is that they exists. In fact, the integer constants and the functions for them we have discussed, were just chosen as a minimal example to write some example programs, like the factorial function.

4 CRSX

CRSX (Combinatory Reduction Systems with Extensions) is a program that can transform expressions, for example λ-expressions. The desired transformation can be specified in the form of a CRS (Combinatory Reduction System). To describe the possibilites of CRSX, we will first have a look at reduction systems in general. After that, the concept of a combinatory reduction system will be explained. Finally, some of the extensions that CRSX offers in addition to CRS will be discussed.

4.1 Abstract reduction systems

An *Abstract reduction system*, or ARS, consists of two things:

- A set of objects, or *terms*. This set may be infinite.
- And a binary relation on the terms, written as an arrow. That is, for some terms a and b, we have $a \to b$. We say b is a "reduction" of a.

Note that no internal structure on the set of terms is assumed, yet, despite their name. This simple concept is enough to introduce some important properties we will use later with CRS. Informal definitions of these will be given, a formal introduction can be found in [Terese, 2003].

Notation

The double headed arrow was already used in Section 2.3. In general, $a \twoheadrightarrow b$ means that a can be reduced to b in zero or more steps. So either a equals b, or there is a number of intermediate terms $x_1 ... x_n$ so that $a \to x_1 \to ... \to x_n \to b$.

When we are using multiple reduction systems on the same set of terms, the arrow gets a subscript to indicate the system it's from. So $a \to_\alpha b$ means "$a \to b$ in reduction system α". In such cases, an arrow without a subscript refers to the union of the discussed systems.

Properties

We will use the following properties of ARS:

Normal form A term a is *reducible* if there exists some b so that $a \to b$. If a is not reducible, it's called a *normal form*. When $a \twoheadrightarrow b$ and b is a normal form, it is called a normal form of a.

Termination An ARS is *terminating* if there exists no inifite sequence of reductions. This means that starting with any term a, after a finite number of reductions, no additional reductions are possible. The resulting term is therefore a normal form of a. So in a terminating ARS, every term has a normal form.

Determinism When every term can be reduced in at most one way, the ARS is called *deterministic*.

Confluence An ARS is *confluent* when the possible reductions from a term can always be reunited: When $a \twoheadrightarrow b_1$ and $a \twoheadrightarrow b_2$, there must exists some c so that $b_1 \twoheadrightarrow c$ and $b_2 \twoheadrightarrow c$.

$$\begin{tikzcd}
& a \arrow[ld, two heads] \arrow[rd, two heads] & \\
b_1 \arrow[rd, two heads, dotted] & & b_2 \arrow[ld, two heads, dotted] \\
& c &
\end{tikzcd}$$

Theorems

The interaction between these properties is captured in theorems about ARS. A few are presented here, with informal proofs. For a more extensive account, we refer again to [Terese, 2003].

- *A deterministic ARS is also confluent.*
  Different possible reductions can be reunited, simply because there is only one possible reduction.

- *In a confluent ARS, each term has at most one normal form.*
  If a and b are normal forms of a term, there must be a c so that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$ by confluence. But because they are normal forms, no further reduction is possible, and c must equal both a and b, so a must equal b.

- *If an ARS is both confluent and terminating, each term has a unique normal form.*
  Consider a term. It has *at least one* normal form because the ARS is terminating, and it has *at most one* normal form because the ARS is confluent.

4.2 λ-calculus as ARS

Both λ-calculus and lazy λ-calculus can be seen as an ARS. Our earlier use of the term *deterministic* coincides with the ARS definition. Some important properties of the calculi can be summed up as follows:

	λ-calculus	lazy λ-calculus
terminating	no	no
deterministic	no	yes
confluent	yes	yes

The confluence of λ-calculus was already mentioned in Section 2.2. The confluence of both calculi will be further investigated in Section 5.3

4.3 Combinatory reduction systems

Combinatory reduction systems, or CRSs, where introduced by J. W. Klop ([Klop, 1980]). A CRS is also an ARS, so the properties and theorems we have for ARSs also apply to CRSs. An ARS puts no limitations on the set of terms, any set will do. The reduction relation, too, can be chosen freely. This is not the case in a CRS: Here, both are constructed in a systematic manner according to a specification. We will first have a look at how the set of terms is constructed, and then how the reduction relation is specified and constructed.

Terms

The terms of a CRS are build of *variables* and *functions*. Functions need to be supplied additional terms as arguments. Following the conventions of CRSX, we will write variables with lower case letters, and functions using upper case. Here are two examples:

1. `F[x]`

2. `F[G, F[G, G]]`

In the first example, clearly `F` is a function, and `x` a variable. The second example also uses a function `F`, but here it is supplied two arguments instead of one. Because of this, it's considered to be a different function than the one used in example 1, even if it has the same name. The second example also uses a function `G`, having zero arguments. In full, it could be written `G[]`, but it is shortened to leave of the brackets. Functions without arguments are called *constants*.

Variables are valid terms on themselves, but can additionally be used to construct an *abstraction*. An abstraction consist of a variable called the parameter, followed by a dot, and then the abstraction body. The body is again a term, and in this term a variable is bound to the parameter if it has the same name. To be precise, it is only bound if it is not already bound by another abstraction inside the body; this is similar to binding in λ-calculus, which is discussed in Section 2.5.

Abstractions usually occur as an argument of a function, and this will in fact be the case in all systems discussed in this thesis. The surrounding function is called a *binder*. Some example terms that use abstractions are:

```
F[x . x]
F[x . H[G]]
H[F[x . F[y . K[x, y]]]]
```

Just like with abstractions in λ-calculus in Section 2.5, bound variables can be renamed if they don't conflict with other variables, the renamed term is considered to be equal to the original. Again, α-conversion may be necesarry during substitutions to avoid name capturing. We do this implicitly (and so does CRSX).

Rules

The reduction relation is constructed from a set of rules. We will begin with a simple subset of rules, and then extend to the full set of rules in four steps.

1: Simple rules Simple rules can be expressed using terms. Such a rule consist of two terms, called pattern and replacement. They are separated with an arrow. An example of such a rule is:

```
F[G, G] → G
```

A rule can be used to reduce any term that contains the pattern as a subterm. This rule for example, can be used to create the following reduction sequence:

```
  F[G, F[G, G]]
→ F[G, G]
→ G
```

2: Rules with variables If the pattern contains one or more variables, it doesn't have to be exactly the same as a subterm to be used. A pattern also *matches* when the variables in the pattern are substituted by other variables. The same substitutions have to be made in the replacement. For example, the rule `F[a]` → `G[G[a]]` can be used to make the reduction `H[F[x]]` → `H[G[G[x]]]`.

3: Rules with meta-variables A variable in a pattern can only match another variable. To allow for more powerful rules, *meta-variables* are introduced. Along with them come *metaterms*, which are terms extended with meta-variables. Again following CRSX convention, meta-variable names will start with a hash sign (`#`), or if only one is needed, just be named `#`. So for example, `F[#]` is a metaterm. Any ordinary term is also a metaterm.

Rules are allowed to use not just terms, but metaterms, for the pattern as well as the replacement. There are two limitations: Firstly, the pattern can not just be a single variable or meta-variable, it has to start with a function. Secondly, in the replacement part only meta-variables are allowed that are also in the pattern; otherwise no substitution can be made for them.

To match a pattern with a subterm, the meta-variables can be substituted by any term, not just a single variable. Again, in the replacement, the same substitutions have to be made. However, the substitute term is not allowed to contain variables that would become bound by an abstraction in the pattern on substitution. For example, the rule:

```
F[#] → G[G[#]]
```

can be use to reduce:

```
H[F[H[x]]] → H[G[G[H[x]]]]
```

The match is made by substituting `H[x]` for `#`.

These meta-variables form a useful tool, but in this definition, they have a problem: they can break variable bindings. For example, this rule:

```
G[v . #] → F[G[v . H], #]
```

could be used to make this reduction:

```
G[x . A[x]] → F[G[x . H], A[x]]    [WRONG]
```

Here, `v` is no longer bound, because it is no longer in the body of the abstraction. This would break α-conversion. The solution is to make some variables "forbidden" for the meta-variable. Specifically, if a pattern contains an abstraction binding variable `v`, and meta-variable `#` is in the body of the abtraction, `v` is forbidden for `#`. This means that during matching, the meta-variable `#` can not be matched with a subterm if that subterm contains the variable that `v` is matched with. In the given example, this means that in the rule, `v` is forbidden for `#`. In the matching, `v` is matched with `x`, so `#` is forbidden from matching a subterm containing `x`. In the example, `#` was matched with `A[x]`, so this was indeed not allowed.

4: Rules with binding Meta-variables We reach the complete definition of CRS by making the meta-variables even more powerful: In the pattern, we allow them to be supplied with a number of bound variables they are sensitive to. When a meta-variable is sensitive to a bound variable, it is no longer "forbidden" for it. Instead, for every occurence of the meta-variable in the replacement part of the rule, a substitution metaterm is supplied for each of the variable it's sensitive to. This additional information for a meta-variable is supplied using the same syntax with which functions are supplied arguments, but the meaning is different.

Let's look at some examples. Here is a rule where a bound variable is substituted by a simple constant:

```
B[x . #[x]] → #[G]
```

Three reductions this rule allows are:

```
    B[a . H[a]] → H[G]
and B[z . F[z, z]] → F[G, G]
and H[B[z . K]] → H[K]
```

A bound variable can also be replaced by another bound variable, or even the term matching a meta-variable:

```
B[x . #[x]] → B[x . #[#[x]]]
```

The tree starting terms of the previous example can now be reduced as:

```
    B[a . H[a]] → B[a . H[H[a]]]
and B[z . F[z, z]] → B[z . F[F[z, z], F[z, z]]]
and H[B[z . K]] → H[B[z . K]]
```

But a meta-variable can match more than one bound variable. For example, the rule:

```
B[x .  B[y . #[y, x]]] → B[z . #[z, K[z]]]
```

can make the reductions

```
    B[r . B[s . H[s]]] → B[z . H[z]]
and B[r . B[s . H[r]]] → B[z . H[K[z]]]
```

Let's take a look at what exacly happens at that last reduction. The pattern contains variables `x` and `y`, and a meta-variable `#`. They all need the proper substitution to match the given term:

- `x` is replaced by `r`
- `y` is replaced by `s`
- `#` is replaced by `H[r]`

The parameters supplied to `#` in the pattern and replacement mean that:

Bound occurrences of `y` *and* `x` *in* `#` *are replaced by* `z` *and* `K[z]`

Due to the earlier substitution, this has become:

Bound occurrences of `s` *and* `r` *in* `H[r]` *are replaced by* `z` *and* `K[z]`

So the substitution value of `#` is `H[K[z]]`. Applying this to the replacement part of the rule gives `B[z . H[K[z]]]`.

CRS Specification

A CRS is specfied in two parts: A set of functions (and constants) to build terms from, and a set of reduction rules. Sometimes it's useful to label the individual rules in a CRS, so it can be made explicit which rule is used for a reduction. A rule name can also refer to the CRS with the same terms, but only the named rule.

4.4 Implementation and extension: CRSX

CRSX is a program that can let a computer do CRS reductions. The software is still under development, led by Kristoffer Rose at IBM Research. It is available at `http://crsx.sourceforge.net/`. Our research was done using version 24.

The program is fed the specification of a CRS and an input term. In it's primary mode of operation, it will then execute reductions steps on the input term until no further reduction can be made; the resulting term is the output. This is similar to how we defined λ-calculus. This means that CRSX, loaded with a CRS specification, can be thought of as a programming language: a program that reduces terms. Nevertheless, there is an important difference: In λ-calculus, we use the fact that the reduction rules are confluent, and in lazy λ-calculus they are even deterministic. But CRSX can be loaded with any CRS, so we have no such guarantees! As said, the input of CRSX can be tought of as *evaluator for a programming language + program in that language to evaluate*, but the more traditional view *program + input to run the program on* is also useful.

Implementation

To specify a CRS in CRSX, the specification of the used functions (and constants) can be skipped. It is simply taken to be the union of all functions used in the CRS and the input term.

Extensions

CRSX offers a number extensions on top of CRS. We discuss only these that we will use. A more complete account can be found in the documentation, in [Rose, 2011], and on the website ([Rose, 2010]).

The discussed extensions use the reserved function name `$`. A number of internal reductions and matchings for this function are done, that transcend the capabilities of CRS.

Integers In λ-calculus, we added the integers as an infinite number of constants. CRS only allows for a finite number of constants to be defined, but CRSX doesn't need this definition. Any unique digit string is also a valid function (or constant) name, so the integers are already in CRSX. To do aritmetic, the `$` function can be used. For example to calculate $2 + 3$, the expression `$[Plus, 2, 3]` is automaitcly reduced to `5`

Matching Extra requirements can be put on a pattern, so that it matches only a subset of the expressions a "pure" CRS pattern would match. Most importantly, `$[NotMatch,` t`,` p`]` matches the same expressions as the pattern p would, except when they also match t.

5 Transforming Haskell using CRSX

Tranforming Haskell using CRSX would consist of three steps: Translating Haskell to lazy λ-calculus, transforming the λ-expression, and translating back to Haskell. Here we concentrate on the transformation of λ-expressions.

5.1 λ-expressions as CRS terms

To transform λ-expressions using CRSX, we need a way to represent λ-expressions as CRS terms. To do this, we define a funtion f that takes a λ-expression, and outputs a CRS term. This is relatively straightforward: in section 3.1 we listed the different forms a λ-expression can take, so we define f for these cases. Here `A` and `B` stand for λ-subexpressions, `v` for a variable, and `C` for a constant.

- An application
  For application, we introduce a new function, with two parameters for the representations of the left and right subexpression. This function will be called `@`.
  f(`A B`) = `@[`f(`A`), f(`B`)`]`

- An abstraction
  For abstraction too, we use a new function, with one parameter. It is called λ. A λ-abstraction is represented by a CRS-abstraction inside the λ function. Note that in CRS terms, just as in λ-calculus, λ is called a *binder*.
  f(λ `v . A`) = λ`[v .` f(`A`)`]`

- A variable
  variables can be copied directly to CRSX
  f(`v`) `= v`

- A constant
  constants can be copied directly to CRSX
  f(`C`) `= C`

- A `Let` contruction
  The `Let` contruction also *binds* a variable. This binder will be called `Let`. Because it binds the variable in two subexpressions, We pair them up with a new function called `In`
  f(`Let v = A In B`) = `Let[v . In[`f(`A`), f(`B`)`]]`

5.2 Alternative notations in CRSX

We just introduced four CRSX function to represent λ-expressions in CRSX. The most important two are `@` and λ. For both, CRSX provides alternative notations, which we will use, not just in terms but also in patterns and replacements:

Application The `@` function was designed to represent application, and when it has two arguments, as in `@[A, B]`, it can be written as just the arguments seperated by a space: `A B`. This way, it looks just like the application in λ-calculus, but it is important to remember the `@` is still there.

Singe-argument binders When a function with one argument contains an abstraction, the brackets of the function can be left off. We will use this only on the λ binder, so this too looks just like λ-calculus.

Let notation We will also write the `Let`-construction in λ-calculus style. This is not actually supported by CRSX, but it makes the CRS rules easier to read.

CRS terms didn't need parentheses for grouping expressions, but these alternative notations can cause ambiguities. So parentheses are used when needed, with the same conventions as in λ-calculus. So the CRS term `A B C` actually stands for `@[@[A, B], C]`.

5.3 Confluence of the λ-calculi

That lazy λ-calculus is confluent, follows trivially from the fact that it is deterministic. For the ordinary λ-calculus, this is not so simple: we mentioned that it is confluent a few times before, be how do we know this?

Another important fact that we will need is the following: If, in λ-calculus, an expression has a normal form that is a constant, the same expression in lazy λ-calculus has the same normal form. Remeber that λ-calculus is not deterministic, so this means there might *also* be an infinite reduction sequence for the expression. But lazy λ-calculus is deterministic, so if there is a normal form, it will arive at it after a finite number of reductions. In other words: *If in λ-calculus there is a right and a wrong way to evalutate an expression, lazy λ-calculus will use the right way.*

These two important theorems were proven for *pure* λ-calculus, see for example [Rosser, 1982]. Unfortunatly, these theorems do not directly carry over to our version of λ-calculus, with the added constants and `Let`-construction. But the theorems *are* valid in a lot of variations of λ-calculus, including ours. A general way to deduce such theorems using CRS, that works on our version as well, is described in [Wells and Muller, 2000].

5.4 Lazy Evaluation using CRSX

Describing β-reduction is very easy with CRSX:

$$(\lambda\ \mathtt{x}\ .\ \#_e\mathtt{[x]})\ \#_v \rightarrow \#_e[\#_v]$$

However, running this system on a λ-expression will perform β-reduction on all possible subexpressions in an unspecified order, just like ordinary λ-calculus. To implement lazy λ-calculus in CRSX, we need a way to give special status to the whole expression, as opposed to any subexpression, because lazy λ-calculus makes a distinction between those. We do this by demanding a *marker* is placed around the expression to be evaluated: Simply place the expression inside the function `E[ ]`. The extended CRS is so designed that it begins the evaluation at this marker.

Here is the algorithm. It is a CRSX implementation of the program in section 3.1. As indicated by the use of the `$` function, it is not a *pure* CRS, because it makes use of some of the extensions CRSX provides on top op CRS. This is only needed to handle the constants we added to λ-calculus: Lazy evaluation for *pure* λ-calculus could be expressed using a *pure* CRS. For simplicity, this version does not handle the `Let`-construction.

```
E[λ x . #[x]]                → λ x . #[x]
E[#_M #_N]                   → App[E[#_M], #_N]
App[λ x . #_M[x], #_N]       → E[#_M[#_N]]
App[C[#_M, P[#_P]], #_N]     → CA[C[#_M, P[#_P]], E[#_N]]
CA[C[#_M, P[#_P]], C[#_N]]   → C[#_M #_N, #_P]
C[#, 0]                      → C[#]
E[Plus]                      → C[Plus, P[P[0]]]
E[Minus]                     → C[Minus, P[P[0]]]
E[Times]                     → C[Times, P[P[0]]]
E[Zero]                      → C[Zero, P[0]]
E[$[IsInteger, #I]]          → C[#I]
C[#_op #_a #_b]              → C[$[#_op, #_a, #_b]]
C[Zero 0]                    → λ x y . x
C[Zero $[NotMatch, 0, #_I]]  → λ x y . y
```

5.5 Studied transformations

We will study three transformations on lazy λ-calculus. While they are quite different from each other, they have some important properties in common:

- They preserve the meaning of the program. That is, if the original program has a normal form under lazy λ-calculus, the transformed program has the same normal form. And if the original does not have a normal form, neither does the transformed.
- They have all been studied as a means of *optimization*: The preprocessing of a program to speed up the evaluation process.

The studied transformations are: *Let-floating*, *Lambda-lifting* and transformation to *Continuation-Passing-Style*.

6 Let-floating

Imagine implementing a function to calculate $a^2 + b^2$ depending on a and b. One possible solution is:

```
λ a b . Let square = λ x . Times x x
        In Plus (square a) (square b)
```

In this implementation, the function is defined in terms of a helper function, `square`. The definition of `square` is attached to the main function using a `Let` construction. But it could equally well be attached at a higher level: to the whole function instead of the body. The resulting definition is:

```
Let square = λ x . Times x x
In λ a b . Plus (square a) (square b)
```

There may not be an immediate advantage to the new version, but the transformation is part of a more complex optimizing transformation, as discussed in [Peyton Jones, 1987].

The transformation of moving a `Let`-construction to a bigger subexpression is called *Let-floating*: when a `Let`-construction is the body of an abstraction, we can “pull” the `Let` “out”. The abstraction ends up in the right part of the `Let`-construction. But there is the risk of breaking a variable binding: the transformation is only possible when the left part of the `Let`-construction does not contain the variable bound by the abstraction.

6.1 Let-floating with CRSX

The let-floating transformation can be implemented in pure CRS with a single rule. We will use the meta-variables `#` and `#`$_e$ to match the left and right part of the `Let`-construction. By making `#` *not* sensitive to the variable bound by the abstraction, the pattern only matches when the left part does not contain that variable, which is precisely when the transformation is possible. In λ-calculus style, the rule reads:

```
      λ x . Let y = #[y] In #ₑ[x, y]
→f
      Let y = #[y] In λ x . #ₑ[x, y]
```

The same rule, without using λ-calculus style:

```
      λ[x . Let[y . In[#[y], #ₑ[x, y]]]]
→f
      Let[y . In[#[y], λ[x . #ₑ[x, y]]]]
```

When CRSX is loaded with this specification, it will apply the rule repeatedly until no more replacements can be made. It will transform the earlier example in two steps:

```
      λ a b . Let square = λ x . Times x x
              In Plus (square a) (square b)
→f
      λ a .   Let square = λ x . Times x x
              In λ b . Plus (square a) (square b)
→f
              Let square = λ x . Times x x
              In λ a b . Plus (square a) (square b)
```

6.2 Correctness

The Let-floating transformation is not supposed to change the meaning of he program: When the program is evaluated using lazy λ-calculus, the result should be the same. We will show this is the case after *one* reduction step. Then from induction it follows the same holds for the complete transformation as executed by CRSX.

Let $\mathtt{P}_A$ be a program that is reduced to $\mathtt{P}_B$ using one let-floating reduction step: $\mathtt{P}_A \rightarrow_f \mathtt{P}_B$. This means there is a subexpression `A` of $\mathtt{P}_A$ that matches the pattern of the let-floating rule, and is replaced. The replacement `B` is a subexpression of $\mathtt{P}_B$.

The pattern and the replacement of the let-floating rule can both be reduced with one `Let`-reduction step from λ-calculus to the same expression:

$$\begin{tikzcd}
\lambda\ \texttt{x . Let y = \#[y] In \#}_e\texttt{[x, y]} \arrow[rr, "f"] \arrow[dr, "L"'] & & \texttt{Let y = \#[y] In } \lambda\ \texttt{x . \#}_e\texttt{[x, y]} \arrow[dl, "L"] \\
& \lambda\ \texttt{x . \#}_e\texttt{[x, Let y = \#[y] In \#[y]]} &
\end{tikzcd}$$

Because the let-floating rule reduces `A` to `B`, there is a substitution for the meta-variables of the rule, so that the pattern equals `A` and the replacement equals `B`. The above diagram is still valid under this substitution:

$$\begin{tikzcd}
\texttt{A} \arrow[rr, "f"] \arrow[dr, "L"'] & & \texttt{B} \arrow[dl, "L"] \\
& \texttt{U} &
\end{tikzcd}$$

Here `U` is used for the common `Let`-reduction of `A` and `B`. `A` and `B` are different subexpression of the otherwise identical expression $\texttt{P}_A$ and $\texttt{P}_B$. We can build $\texttt{P}_U$ from `U` in the same way. The `Let`-reductions are part of λ-calculus. We'll call the λ-calculus reduction system λ. In λ, a reduction can be done on a subexpression, independent of what bigger expression it is part of. The same holds true for any CRS, including f. So for the entire programs we can make this diagram:

$$\begin{tikzcd}
\texttt{P}_A \arrow[rr, "f"] \arrow[dr, "\lambda"'] & & \texttt{P}_B \arrow[dl, "\lambda"] \\
& \texttt{P}_U &
\end{tikzcd}$$

If $\texttt{P}_A$ is a valid program, it either reduces to a constant `C` in lazy λ-calculus, or it does not reduce at all. Let's investigate the first case. `C` is a normal form of $\texttt{P}_A$ in lazy λ-calculus, so it is in ordinary λ-calculus as well. Because λ-calculus is confluent, as we saw in Section 5.3, `C` is also a normal form of $\texttt{P}_U$, and of $\texttt{P}_B$ too because $\texttt{P}_B$ is a λ-reduction of $\texttt{P}_U$:

$$\begin{tikzcd}
\texttt{P}_A \arrow[rr, "f"] \arrow[dr, "\lambda"] \arrow[dd, two heads, "\lambda"'] & & \texttt{P}_B \arrow[dl, "\lambda"] \\
& \texttt{P}_U \arrow[dl, dotted, two heads, "\lambda"] & \\
\texttt{C} & &
\end{tikzcd}$$

Because `C` is a normal form of $\texttt{P}_B$, using the second theorem of Section 5.3, we conclude that using lazy λ-calculus, $\texttt{P}_B$ will evaluate to `C` just like $\texttt{P}_A$. In the case $\texttt{P}_A$ does not reduce, it does not have a normal form in ordinary λ-calculus. This means $\texttt{P}_U$ can not have a normal form either, and again using confluence, $\texttt{P}_B$ doesn't have one, so it can not be reduced using lazy λ-calculus. So we have proven lazy evaluation will yield the same result for $\texttt{P}_A$ and $\texttt{P}_B$.

7 Lambda-lifting

A good way to optimize a λ-calculus program is to convert it to *supercombinators.* The transformation itself is called *lambda-lifting.* They are discussed in detail in [Peyton Jones, 1987]. We give a summary of the relevant material.

7.1 Supercombinators

A supercombinator is a λ-expression of the form

```
λ v₁ v₂ ... vₙ . #
```

with the following properties:

- The expression can start with any number of abstractions ($n \geq 0$).
- `#` is not an abstraction.
- `#` contains no variables that are bound outside the supercombinator.
- Abstractions in `#` that are not inside other abstractions in `#` are supercombinators too.

The process of β-reduction can be optimized in a program that is a supercombinator: it can be shown that β-reduction can be delayed until the expression is supplied n arguments. With the right setup, they can then be substituted *all at once.* The constants `Plus` and `Times` already behave like supercombinators: There is no need to evaluate them until they have two arguments. When they have two, the reduction is simple.

7.2 Lambda-lifting

Lambda-lifting can transform any expression into a supercombinator, as long as there are no variables in it that are bound outside the expression. This includes a whole program. We will see how this works using one of the examples from [Peyton Jones, 1987]:

$\texttt{(}\lambda\texttt{ x . (}\lambda\texttt{ y . Plus y }\underline{\texttt{x}}\texttt{) x) 4}$

In this expression, the inner abstraction is not a supercombinator, because it contains the variable `x` (underlined in the expression), which is bound outside it. The outer abstraction is not a supercombinator either, because its body contains an abstraction that is not. The underlined variable `x` is the problem here: the binding to it's abstraction passes through another abstraction. The solution is to "cut" the binding in two: Replace the subexpression

```
(λ y . Plus y x)
```

by

$\texttt{(}\lambda\texttt{ x}_2\texttt{ y . Plus y x}_2\texttt{) x}$

That this new subexpression doesn't change the meaning can be seen by the fact that it can be converted back with one β-reduction. The entire new expression becomes:

```
(λ x . (λ x₂ y . Plus y x₂) x x) 4
```

The variable $\texttt{x}_2$ still refers to the abstraction of `x`, but indirectly: they are seperate variables, but we have ensured that $\texttt{x}_2$ will get the value of `x`. The abstraction of $\texttt{x}_2$ is now a supercombinator. The outer abstraction is too, because the inner is, and in fact the whole program is a supercombinator.

7.3 Lambda-lifting with CRSX

Implementing Lambda-lifting in CRSX is not a trivial task. We found a way to do this, the CRS is included in Appendix A, but it is not very readable. One problem is the decision to lift or not lift a variable. This choice depends on the number of groups of abstractions surrounding it within the scope of the variable. CRSs are not well prepared to handle such queries.

Our solution is to lift *every* variable, then undo those transformations that are not necessary. In the example, the lifting transformation can be reversed by using β-reduction on the new variable. But if multiple variables are lifted, *generalized β-reduction* is required: If n nested abstractions are supplied n variables, β-reduction can be used to substitute them one by one, from left to right. It is valid to skip some substitutions. The next substitution can then not be described by β-reduction, and requires generalized β-reduction.

To execute a generalized β-reduction, the argument needs to be moved down through a number of applications, and then the same number of abstractions, to reach the abstraction it matches with. In a CRS, this has to be done step by step, using functions to mark the expressions.

To decide if a generalized β-reduction is possible without braking the supercombinator structure, information about the surrounding abstrations has to be collected at every variable. Again, this has to be done step by step.

These step by step actions are what make the CRS hard to read. The idea is simple: a subexpression has to be moved to another location in the expression. We have a clear idea of where this new location is, but this idea cannot be translated to the language of CRSs. The relocation has to pass through every subexpression on the way, every step described.

8 Continuation Passing Style

A function is in *Continuation Passing Style*, or *CPS*, if the function takes an extra parameter that represents what to do with the result. In stead of returning the result, it's applied to this extra parameter, called *continuation.* In [Plotkin, 1975], which also gives more details on CPS, Plotkin describes how a λ-expression can be converted to CPS entirely. This involves not only converting the abstractions, but the applications as well: When a function does not return it's result, but expects an extra parameter to apply the result to, calling a function is different as well. The conversion we'll use is designed for a lazy λ-calculus with constants, just like ours. It does not have the `Let`-construction.

8.1 CPS with CRSX

The conversion Plotkin presents can be translated into CRS directly. We introduce two extra functions: `Start` and `C`. The parameter of `C` will be an ordinary (unconverted) λ subexpression. `C` can be placed in a CPS expression: it is a marker indicating that the expression inside still needs to be converted. To run the conversion, the expression needs to be placed in `Start`. It will use `C` to convert the expression. Because the result will be in CPS, it needs a continuation as argument to indicate what to do with the result. `Start` applies `(λ x . x)` to it, this will simply return the result. Because `(λ x . x)` is a common expression, we abbreviate it as `I`.

```
Start[#]            → C[#] I
C[x]                → x
C[$[IsInteger, #]]  → λ κ . κ #
C[Zero]             → λ κ . κ (λ α . Zero (α I))
C[Plus]             → λ κ .
                        κ (λ x α . α (λ y . Plus (x I) (y I)))
C[λ x . #[x]]       → λ κ . κ (λ x . C[#[x]])
C[#1 #2]            → λ κ . C[#1] (λ α . α C[#2] κ)
```

The reductions will introduce new `C` markers; they will always contain a λ-expression. The reduction rules will convert a `C` depending on the form of the λ subexpression inside it: there is a rule for every form. This means that a `C` expression can always be reduced. Since CRSX only stops when no more reductions can be made, the output will not contain any `C` markers anymore, provided that the input is a valid λ-expression.

The resulting expression is in continuation passing style, so it is assumed the functional constants are as well. So for example `Plus` is no longer a function of two arguments, but of three: a continuation and two integers. The sum of the integers will be applied to the continuation.

8.2 Correctness

We were able to prove the correctness of the let-floating CRS using CRS theorems. This may also be the case for the continuation passing style CRS. Plotkin provides a correctness proof of the transformation in his paper, but the methods used fall outside the CRS framework. If a proof based solely on CRS methods exists, we do not know.

9 Conclusion

CRSs nicely match the properties on λ-calculus: The concept of abstractions binding variables is handled well. We have seen how three transformations on lazy λ-calculus can be implemented using CRSX. The results were mixed:

Let-floating

Let-floating can be elegantly implemented, using pure CRS. The CRS specification has only one rule, that can be said to reflect the concept of let-floating clearly.

Lambda-lifting

Implementing lambda-lifting is a different story. It seems that a CRS is not the right tool for this transformation. That it is nevertheless possible, is not surprising: The fact that we also build a lazy λ-calculus evaluator shows that we could in theory implement any transformation that a computer can do. But instead of building an evaluator, it might be better to program the transformation in a λ-calculus based programming language directly. One possible choice is Haskell.

Continuation Passing Style

The transformation to CPS was implemented with ease; the CRS is almost identical to description given by Plotkin in [Plotkin, 1975]. This does not mean CRSX is the best choice to implement it. The CRS looks like a functional program, and in fact it is. An implementation in Haskell, for example, would *also* look exactly like the original description. Nevertheless, CRSX proved to be a useful tool here.

Other transformations

One kind of transformations we have not studied are optimizations based on equivalence of constants. A simple example is the following CRS:

```
Plus 0 # → #
```

The scope of such transformations can be extended from constants to commonly used functions, for example those in the standard library of a programming language. CRS could prove a useful tool for equivalence of functions, because it naturally handles bound variables.

Suitability for implementation

Even if CRSX is only suitable for the implementation of *some* transformations, it is very powerful when it is: Transformations that can convert arbitrary subexpression, without regard for the order of transformations, benefit from the fact that CRS does this naturally. The handling of bound variables is also elegant: implementations in other systems would probably need an explicit mechanism to prevent name capture, for example using α-conversion. This too, CRS does naturally.

Suitability beyond implementation

In the case of let-floating, the CRS provided a framework to prove the transformation does not change the evaluation result. The technique we used can, and is, generalized to powerful theorems; [Terese, 2003] is a good source. The use of CRS might also be a good way to step up to the *automated* proving of desirable properties of transformations. We conclude that CRS is an interesting system for transformations of functional programming languages, and CRSX is a useful tool.

References

[Klop, 1980] Klop, J. W. (1980). Combinatory reduction systems.

[Peyton Jones et al., 2003] Peyton Jones, S. et al. (2003). The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255. http://www.haskell.org/definition/.

[Peyton Jones, 1987] Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Plotkin, 1975] Plotkin, G. (1975). Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125 – 159.

[Rose, 2010] Rose, K. H. (2010). CRSX - combinatory reduction systems with extensions. http://crsx.sourceforge.net/.

[Rose, 2011] Rose, K. H. (2011). CRSX—combinatory reduction systems with extensions. In Schmidt-Schauß, M., editor, *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Rosser, 1982] Rosser, J. B. (1982). Highlights of the history of the lambda-calculus. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, LFP '82, pages 216–225, New York, NY, USA. ACM.

[Terese, 2003] Terese (2003). *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

[Wells and Muller, 2000] Wells, J. B. and Muller, R. (2000). Standardization and evaluation in combinatory reduction systems.

A CRS for lambda-lifting

Lift[#]	→	Top[Down[#]]
Down[L[x . #[x]]]	→	L[x . Down[#[Var[x]]]]
Down[#$_1$ #$_2$]	→	(Down[#$_1$] Down[#$_2$])
Down[Var[#]]	→	M[P[L x . Body[x]] #]
Down[c]	→	CAF[c]
(CAF[#$_1$] CAF[#$_2$])	→	CAF[#$_1$ #$_2$]
(CAF[#CAF] M[#])	→	M[C1[#CAF, #]]
C1[#CAF, #$_1$ #$_2$]	→	(C1[#CAF, #$_1$] #$_2$)
C1[#CAF, P[#]]	→	P[C1[#CAF, #]]
C1[#CAF, L x . #[x]]	→	(L x . C1[#CAF, #[x]])
C1[#CAF, Body[#]]	→	Body[#CAF #]
(M[#] CAF[#CAF])	→	M[C2[#CAF, #]]
C2[#CAF, #$_1$ #$_2$]	→	(C2[#CAF, #$_1$] #$_2$)
C2[#CAF, P[#]]	→	P[C2[#CAF, #]]
C2[#CAF, L x . #[x]]	→	(L x . C2[#CAF, #[x]])
C2[#CAF, Body[#]]	→	Body[# #CAF]
(M[#$_1$] M[#$_2$])	→	M[A1b[#$_1$, #$_2$]]
A1b[#$_t$ #$_u$, #]	→	(A1b[#$_t$, A1c[#$_u$, #]])
A1b[P[#$_t$], #]	→	A1[P[#$_t$], #]
A1c[#$_v$, # \$[NotMatch,#$_v$,#$_w$]]	→	(A1c[#$_v$, #] #$_w$)
A1c[#$_v$, # #$_v$]	→	(A1c[#$_v$, #] #$_v$)
A1c[#$_t$, P[#]]	→	(P[#] #$_t$)
A1[#$_t$, #$_1$ #$_2$]	→	(A1[#$_t$, #$_1$] #$_2$)
A1[#$_t$, P[#]]	→	A2[#, #$_t$]
A2[#$_t$, #$_1$ #$_2$]	→	(A2[#$_t$, #$_1$] #$_2$)
A2[#$_t$, P[#]]	→	P[A2[#$_t$, #]]
A2[#$_t$, L x . #[x]]	→	(L x . A2[#$_t$, #[x]])
A2[#$_t$, Body[#]]	→	A3[Body[#], #$_t$]
A3[#$_t$, L x . #[x]]	→	(L x . A3[#$_t$, #[x]])
A3[Body[#$_1$], Body[#$_2$]]	→	Body[#$_1$ #$_2$]
GBeta[#$_s$, #$_t$ Go[#$_v$]]	→	GBeta[(Go[#$_v$]; #$_s$), #$_t$]
GBeta[#$_s$, #$_t$ v]	→	(GBeta[(Skip;#$_s$), #$_t$] v)
GBeta[#$_s$, P[#]]	→	P[GBeta2[#$_s$, #]]
GBeta2[(Skip; #$_s$), L x . #[x]]	→	(L x . GBeta2[#$_s$, #[x]])
GBeta2[(Go[#$_v$]; #$_s$), L x . #[x]]	→	GBeta2[#$_s$, #[#$_v$]]
GBeta2[(), Body[#$_t$]]	→	Body[#$_t$]
A5[#$_s$, L x . #[x] #$_v$]	→	(A5[(Skip; #$_s$), L x . #[x]] #$_v$)
A5[#$_s$, L x . P[#[x]]]	→	P[A6[#$_s$, L x . #[x]]]
A6[(Skip; #$_s$), L x y . #[x, y]]	→	(L y . A6[#$_s$, L x . #[x, y]])
A6[(), L x . Body[#[x]]]	→	Body[L x . #[x]]
(L y . M[#[y]])	→	(L y . N[#[y]])
(L y . N[#[y]])	→	N[A5[(), L y . GBeta[(), #[Go[y]]]]]
(#$_t$ N[#])	→	(#$_t$ Make[#])
(N[#] #$_t$)	→	(Make[#] #$_t$)
Top[N[#]]	→	Top[Make[#]]
Top[M[P[Body[#]]]]	→	#
Make[#]	→	M[Make[(), #]]
Make[#$_s$, # #$_v$]	→	(Make[(Skip;#$_s$), #] #$_v$)
Make[#$_s$, P[#]]	→	P[Make2[#$_s$, (), #]]
Make2[(Skip; #$_s$), #$_s$2, #]	→	(L w . Make2[#$_s$, (w; #$_s$2), #])
Make2[(), #$_s$, #]	→	Body[Make3[#$_s$, #]]
Make3[(#$_v$;#$_s$), #]	→	(Make3[#$_s$, #] #$_v$)
Make3[(), #]	→	SC[UnBody[#]]
UnBody[Body[#]]	→	#
UnBody[L x . #[x]]	→	(L x . UnBody[#[x]])