On Meaning Preservation of a Calculus of Records

Emily Christiansen and Elena Machkasova
Computer Science Discipline
University of Minnesota, Morris
Morris, MN 56267
chri1101, elenam@morris.umn.edu

Abstract

This paper focuses on meaning preservation of transformations on a system of mutually dependent program components, which models modular programming. Collections of such components are called records. The term *program transformations* refers to changes that compilers and similar tools make to a program to improve its performance and/or readability. Such transformations are *meaning-preserving* if they preserve the behavior of the program. While commonly performed transformations are well tested and widely believed to be meaning-preserving, precise formal proofs of meaning preservation are tedious and rarely done in practice. Optimized programs may have unexpected changes in their behavior due to optimizations. Formal approaches to meaning preservation could lead to more reliable software without sacrificing the program's efficiency. In this paper, we give a formal system for describing program modules and prove some of its important properties.

Records represent collections of components which may depend on each other, with possible mutual recursion. Records can be *evaluated* or *reduced* (i.e. optimized). Evaluation steps model the way the program would be evaluated. Transformation steps happen only during optimization. In this paper we introduce the necessary formal framework and prove two important properties: *confluence of evaluation* and that *a non-evaluation step preserves the state of a term.* Confluence of evaluation means that the result of evaluation of a record does not depend on a specific order of evaluation. The state of a term shows whether the term can be evaluated, and in the case that it can not be evaluated further, what value it has. Confluence of evaluation and preserving the state of a term are necessary fundamental properties for proving meaning preservation of the system of records.

1 Introduction

When programmers write code they usually try to do so in a way that is simple and easy to read. This involves refactoring, which means separating program components into smaller modules or classes with many small functions. With the overhead of handling multiple classes and functions, it is no surprise that there is a trade-off between readability and running time. To alleviate this potential slow down, the compiler should undo most of the spreading out and optimize the code through a series of program transformations that do not change the program's behavior in any way. We call such transformations *meaning-preserving.* Such transformations may include function inlining (replacing a function call by the code of that function), dead code elimination (removing unused variables and code fragments that never execute), and other optimizations performed either at compile-time or at run-time. Knowing that such transformations preserve the meaning of the program is important because the correctness of software relies not only on the program itself but also on the correctness of optimizations made by the compiler. This has significant implications for the writers of safety-critical applications such as programs used by air-traffic controllers or nuclear power plant monitors.

Many language features and properties can be described in language-independent formal systems such as the lambda calculus. Many transformations, especially the newer ones, are not yet described and proven meaning-preserving in a formal system. Formalizing real-life language features and program transformations allows for formal and, ideally, automated proofs of various safety properties, increasing software reliability. In addition to proving meaning preservation of program transformations, such formal systems can be used for proving other properties, such as a guarantee that a program is not going to exceed a certain memory limit or a function stack limit, that a certain variable has a value that stays within a certain range, etc. Currently there exist systems that perform automated verification or proofs of certain properties for a fairly narrow set of specific systems. However, constructing more general systems for proving program properties, especially meaning preservation of optimizations, is still quite challenging.

In this research project we sought to develop a system for representing collections of mutually-recursive components evaluated using a *call-by-name* strategy (see section 2.1 for details). This system is based on and is similar to the lambda calculus. However, for simplicity in this paper we present the system and examples in a more traditional function notation. Our goal was to develop a system in which certain kinds of transformations, namely pre-evaluating program components at compile time, are guaranteed to be meaning-preserving. For our system we have proven the basic properties that are necessary for such transformations to be meaning-preserving. These properties can be used as a base case for an inductive proof of meaning preservation of the entire system. Completing the proof of meaning preservation is our future work.

In the following sections we introduce the calculus of records, explain what meaning preservation means in a formal sense, and define the basic properties that are needed for meaning preservation. We show a few examples that illustrate the basic properties. Proofs of these properties require more complex notations and are not shown in this paper, they are given in [2].

2 Calculus of Records and Meaning Preservation

2.1 Calculus of Records

A calculus is a formal system for modeling data and computations. It is used in the area of computer science called semantics of programming languages to model program elements (function calls, loops, etc.), just like mathematical calculus can be used for modeling physical laws. In this work we describe a calculus that we developed to model collections of components, such as Java or C++ classes, libraries of functions, etc. In our calculus we model these features by records. We use this calculus to describe and study optimizations of different components and interactions of these optimizations.

To understand records, simply think of a class with methods and variables, each method and variable with its own name. A class can be modeled by a record which is an unordered collection of named components that can reference each other, including a mutually recursive dependency. A class can be used by connecting it to some other class through inheritance or a library. Therefore components of a record may be referenced from outside the record. At this point we do not model private components of a class, so any component may be referenced from outside the record.

As an example, consider a record:

$$[n \mapsto 2, f(x) \mapsto x + 3, m \mapsto f(n)]$$

Here the component with the name n has a value 2, the second component represents a function $f(x) = x + 3$, and the last component is an application of the function f to n. The symbol $\mapsto$ binds a name to its definition. For instance, n is defined as 2, and $f(x)$ as $x + 3$. In practice the function and its application can belong to different classes, and the above record may be the result of linking together several classes. Note that in this example the component m depends on both n and $f(x)$, the other two component do not depend on any other components.

The next example illustrates a possibility of mutual dependencies in records:

$$\begin{aligned}&[isOdd(x) \mapsto \texttt{ if } (x == 0) \texttt{ then false else not } isEven(x-1),\\ &isEven(x) \mapsto \texttt{ if } (x == 0) \texttt{ then true else not } isOdd(x-1),\\ &b \mapsto isOdd(2)]\end{aligned} \tag{1}$$

This record has two mutually recursive functions and a component b which is bound to the result of applying one of the functions to a number 2.

The BNF (Backus-Naur form) definition of records given on Figure 1 has two levels: the first one defines individual components, also called *terms*, the second level names these components and groups them into records. We use M to range over terms and D to range over records. One kind of terms is a constant, denoted by c. We use only booleans and integer numbers in our examples, but more constants may be added as needed. Likewise, we use only addition as an operation on integers, other operations can be trivially added.

Additionally, terms may contain a function applied to an argument (denoted by $M_1(M_2)$), an if/then/else statements, or a special term called a *black hole*, denoted $\bullet$, which will be explained in section 2.2.

A record has one or more components surrounded by square brackets and separated by commas. Component names in a record are called *labels*. Since some of the components are functions, function parameters are a part of the label for that function. For instance, the component label $isEven(x)$ contains a function parameter x. We use $L \mapsto M$ to indicate that a label L is bound to a term M . For instance, in example (1) the label $isEven(x)$ is bound to the term `if` $(x == 0)$ `then true else not` $isOdd(x - 1)$, and the label b is bound to the constant `false` . The order of components in a record does not matter.

$$
\begin{array}{lcl}
c & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid 1 \mid 2 \mid ... \\
M & ::= & c \mid x \mid L \mid M_1(M_2) \mid M_1 + M_2 \mid M_1 == M_2 \mid \texttt{ if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 \mid \bullet \\
L & ::= & l \mid l(x_1, ..., x_n) \\
D & ::= & [L_1 \mapsto M_1, \ldots, L_n \mapsto M_n]
\end{array}
$$

Figure 1: Definition of Calculus of Records.

2.2 Evaluation and Transformation Steps

The calculus of records that we have designed describes records, a way records are evaluated, and transformations that can be done on records as optimizations. We describe *reduction* - a step that simplifies (i.e. reduces) a record. The reduction follows the call-by-name strategy explained in section 2.2.1. There may be several ways in which a record can be simplified, some of which may reduce the same component in different ways.

2.2.1 Call-by-name Reduction Strategy

Reductions in our calculus follow a call-by-name reduction strategy. According to this strategy, parameters of a function are passed to the function unevaluated. For instance, if we have a function

```
int f (int x) {
  return x + x;
}
```

If we call this function with a parameter 2 + 3, as in

```
int y = f(2 + 3);
```

then the expression 2 + 3 is copied into x unevaluated, so the return statement effectively becomes `return (2 + 3) + (2 + 3)`. Most programming languages use a more familiar call-by-value strategy, when the expression (in this case 2 + 3) is first evaluated, and then the resulting value is passed to the function. In our example under the call-by-value strategy the return statement immediately turns into `return 5 + 5`.

While in most cases the two reduction strategies produce the same result, there are some cases when the call-by-name strategy makes programs behave differently than the call-by-value. For instance, suppose the function above is called with an argument that itself is a

function call: `f(g(2))`. If the function `g` modifies a global variable (say, adds one to some global counter), then call-by-name strategy will replace the return statement by `return g(2) + g(2)`, and the call to `g` will be made twice, hence the global counter will be incremented twice. On the other hand, the call-by-value strategy will call `g(2)` only once when the value of the parameter is evaluated and pass the result to the function. Therefore the global counter will be incremented only once.

The record calculus applies the call-by-name strategy not just to function applications, but also to substitution of record components. See sections 2.2.3 and 2.2.4 for details.

2.2.2 Evaluation and Non-evaluation Steps

There are two kinds of reduction steps we use to transformation components in our calculus. The first kind models the program execution - these are steps that would be taken to evaluate the program. Since our programs are records, i.e. consist of several components, the order of evaluating components does not matter. However, at any given moment there is only next element that needs to be evaluated in each component. The rules of evaluation are given below:

1. if a component is an addition of two terms or a comparison of two terms (using $==$), then its operands are evaluated left to right, i.e. the evaluation of the second operand does not start until the first one is evaluated.

2. if a component is a function applied to an argument, the argument immediately gets substituted into the function.

3. for an `if` / `else` statement, the condition gets evaluated first, and then the `if` or the `else` branch gets evaluated, depending on whether the condition is true or false.

If a record D_1 evaluates to a record D_2, we write this as $D_1 \Rightarrow D_2$.

For example, the following steps are all evaluation steps:

$$[f(x) \mapsto x + 2, n \mapsto f(3 + 5)] \Rightarrow$$
$$[f(x) \mapsto x + 2n \mapsto 8 + 2] \Rightarrow$$
$$[f(x) \mapsto x + 2, n \mapsto 3 + 5 + 2] \Rightarrow$$
$$[f(x) \mapsto x + 2, n \mapsto 10]$$

The last record in the sequence is completely evaluated.

A detailed example of evaluating `if` / `else` statement is given in section 2.2.4.

Note that the evaluation rules specify only the order in which an individual component is reduced. Different components in a record may be evaluated in any order. For instance, the record $[n \mapsto 2 + 3, m \mapsto$ `if true then` 3 `else` $5]$ is evaluated in one step to $[n \mapsto 5, m \mapsto$ `if true then` 3 `else` $5]$ if we evaluate the component with the label n, and to $[n \mapsto 2 + 3, m \mapsto 3]$ if we choose to evaluate the one bound to m.

Evaluation steps may also involve substitution if a component references a label and needs the result of that label to continue execution. For instance, evaluating the component bound to n in the record below requires substitution from a component bound to m:

$$[n \mapsto m + 2, m \mapsto 3 + 5] \Rightarrow$$
$$[n \mapsto 3 + 5 + 2, m \mapsto 3 + 5]$$

Evaluation defines the way the program will be evaluated at run-time. However, our calculus also models transformation steps that can be performed by the compiler to optimize the record before it gets evaluated. These steps are called *reduction steps* and are denoted $\rightarrow$.

Note that the compiler may perform some of the same steps that the evaluation would perform. Therefor, any evaluation step is a reduction step, which more formally is stated as $\Rightarrow \subset \rightarrow$. There are also reduction steps that are not evaluation steps. We call them *non-evaluation* steps and denote $\hookrightarrow$. The definition of $\Rightarrow$ implies that $\rightarrow = \Rightarrow \cup \hookrightarrow$.

Non-evaluation steps are the same step as evaluation steps, except they are performed "out of turn", i.e. on the elements of the term that *do not* satisfy the evaluation rules above.

For instance, the following reduction is a non-evaluation step since it reduces the argument of a function before applying the function. This contradicts the second rule defining evaluation.

$$[f(x) \mapsto x + 2n \mapsto f(3+5)] \hookrightarrow [f(x) \mapsto x + 2n \mapsto f(8)]$$

This example of non-evaluation reduces the `if` branch instead of reducing the entire component to $2 + 4$, as required by rule 3 of evaluation definition.

$$[x \mapsto \texttt{ if true then } 2+4 \texttt{ else } 3+5] \hookrightarrow$$
$$[x \mapsto \texttt{ if true then } 6 \texttt{ else } 3+5]$$

2.2.3 Term Reduction

A *term reduction* is a reduction that operates on an individual component, without interacting with other components. If the element of a term that gets reduced satisfies one of the three evaluation rules in section 2.2.2 then the term reduction is an evaluation step $\Rightarrow$, otherwise it's a non-evaluation step $\hookrightarrow$. Recall that in either case it is a reduction step $\rightarrow$ since $\rightarrow = \Rightarrow \cup \hookrightarrow$.

For example:

$$[n \mapsto 2 + 3] \Rightarrow [n \mapsto 5]$$

is an evaluation step, whereas this reduction

$$[n \mapsto (3+5) + (1+4)] \Rightarrow [n \mapsto (3+5) + 5]$$

is not, since evaluation would reduce the first operand of the plus (3+5) before the second one.

A term reduction can also apply a function to an argument, evaluate a condition of `if` / `else` , or reduce the entire `if` / `else` statement if its condition is already evaluated. As for the arithmetic operations, if the part of the term that gets reduced satisfies the evaluation rules in section 2.2.2 then the reduction is an evaluation step, otherwise it is a non-evaluation step.

2.2.4 Substitution

Programs may also require the substitution of certain components. We could have a situation like this:

$$[n \mapsto m + 2, m \mapsto 5]$$

Naturally, we would replace m in the first equation with 5, since m equals 5 to get:

$$[n \mapsto 5 + 2, m \mapsto 5]$$

Substitution allows us to substitute the name of a component by a term that the name is bound to, as in the case above. The substitution is call-by-name, just like the term reduction is. For instance, an unevaluated expression $3 + 2$ is substituted for m here:

$$[n \mapsto m + 4, m \mapsto 3 + 2] \rightarrow [n \mapsto m + 4, m \mapsto 3 + 2]$$

Substitution is also not limited to simple arithmetic operations. If a label refers to a function, the function definition is substituted for the label, with the formal parameter substituted by the actual parameter.

Notice that, similarly to the term reduction, if the substitution happens in a position defined by the three evaluation rules then it is an evaluation step, otherwise it is a a non-evaluation step.

The example below is a sequence of evaluation steps (both substitution and term reduction) evaluating the record given in example(1) in section 2.1. In the component b, the function $isOdd$ is called with a parameter 2. Then $isOdd(2)$ calls $isEven(1)$, which in turn calls $isOdd(0)$, at which point `false` is returned. `false` is the negated twice in the returning calls of $isEven(1)$ and $isOdd(2)$, and the resulting value (also `false`) becomes the new value of the component b. Below is the evaluation sequence:

$[isOdd(x) \mapsto$ `if` $(x == 0)$ `then false else not` $isEven(x-1)$,
$isEven(x) \mapsto$ `if` $(x == 0)$ `then true else not` $isOdd(x-1)$,
$b \mapsto$ `if` $(2 == 0)$ `then false else not` $isEven(1)] \Rightarrow$

$[isOdd(x) \mapsto$ `if` $(x == 0)$ `then false else not` $isEven(x-1)$,
$isEven(x) \mapsto$ `if` $(x == 0)$ `then true else not` $isOdd(x-1)$,
$b \mapsto$ `not` $isEven(1)] \Rightarrow$

$[isOdd(x) \mapsto$ `if` $(x == 0)$ `then false else not` $isEven(x-1)$,
$isEven(x) \mapsto$ `if` $(x == 0)$ `then true else not` $isOdd(x-1)$,
$b \mapsto$ `not if` $(1 ==)$ `then true else not` $isEven(1)] \Rightarrow$

$[isOdd(x) \mapsto$ `if` $(x == 0)$ `then false else not` $isEven(x-1)$,
$isEven(x) \mapsto$ `if` $(x == 0)$ `then true else not` $isOdd(x-1)$,
$b \mapsto$ `not not true` $] \Rightarrow$

$[isOdd(x) \mapsto$ `if` $(x == 0)$ `then false else not` $isEven(x-1)$,
$isEven(x) \mapsto$ `if` $(x == 0)$ `then true else not` $isOdd(x-1)$,
$b \mapsto$ `true` $]$

Note that the function definitions stay the same and may be used again if the record is linked with another record that calls $isOdd$ or $isEven$.

The following example illustrates a non-evaluation substitution step:

$$[m \mapsto \texttt{ if true then } 5 \texttt{ else } n, n \mapsto 4+3] \hookrightarrow$$
$$[m \mapsto \texttt{ if true then } 5 \texttt{ else } 4+3, n \mapsto 4+3]$$

2.2.5 Black Holes

Sometimes components depend on each other such that it cannot be meaningfully resolved. Take for example the record: $[x \mapsto y+5, y \mapsto x]$. If we tried to evaluate the component y, we would quickly see that it is impossible:

$$[x \mapsto y+5, y \mapsto x] \Rightarrow$$
$$[x \mapsto y+5, y \mapsto y+5] \Rightarrow$$
$$[x \mapsto y+5, y \mapsto y+5+5] \Rightarrow \cdots$$

The component y has an infinite number of 5s. The components in this system of equations rely on each other such that they cannot be meaningfully resolved. In our calculus instead of an infinite evaluation, the component evaluates in one step to a special symbol $\bullet$ called a *black hole*:

$$[x \mapsto y+5, y \mapsto x] \Rightarrow \tag{2}$$
$$[x \mapsto y+5, y \mapsto \bullet]$$

Black holes can happen for two reasons in our calculus. The first is where a given label depends on itself, such as in the above example or in the record $x \mapsto x$.

The second reason is that one component references another component bound to a black hole so that the label is in the position that needs to be evaluated next. For instance, continuing example (2) above, we get

$$[x \mapsto y+5, y \mapsto \bullet] \Rightarrow [x \mapsto \bullet, y \mapsto \bullet]$$

Reduction to a black hole is considered to be an evaluation step only. The reason for that is that a black hole may appear in a component as a result of substitution, but it does not necessarily imply that the component can not be meaningfully resolved. Consider the following example (here the substitution is a non-evaluation step):

It is important to note that not all components that depend on themselves will go to black holes. Consider this example:

$$[n \mapsto \bullet, m \mapsto \texttt{ if } (2>3) \texttt{ then } n \texttt{ else } 5] \quad \hookrightarrow$$
$$[n \mapsto \bullet, m \mapsto \texttt{ if } (2>3) \texttt{ then } \bullet \texttt{ else } 5] \quad \Rightarrow$$
$$[n \mapsto \bullet, m \mapsto 5]$$

Likewise a component may depend on itself in a way that does not make it into a black hole (here y is another component of the record):

$$[x \mapsto \texttt{ if } (y == 3) \texttt{ then } x \texttt{ else } y]$$

The notion of a black hole was first introduced by Ariola and Klop in [1].

3 Meaning Preservation

3.1 Meaning of a Record

A compiler transformation is meaning preserving if it does not change the behavior of a program. In our calculus the behavior of a record is defined by the result of its evaluation. For instance, the record

$$[isOdd(x) \mapsto \texttt{ if } (x == 0) \texttt{ then false else not } isEven(x-1),$$
$$isEven(x) \mapsto \texttt{ if } (x == 0) \texttt{ then true else not } isOdd(x-1),$$
$$b \mapsto isOdd(2)]$$

evaluates to

$$[isOdd(x) \mapsto \texttt{ if } (x == 0) \texttt{ then false else not } isEven(x-1),$$
$$isEven(x) \mapsto \texttt{ if } (x == 0) \texttt{ then true else not } isOdd(x-1),$$
$$b \mapsto \texttt{ false }],$$

as shown in section 2.2.4. Therefore, the meaning of this record is that its components $isOdd(x)$ and $isEven(x)$ are functions ($isOdd(x)$ returns true when its parameter is odd, and false otherwise, and $isEven(x)$ returns true and false the other way around), and the component b is false.

For a calculus to be meaning preserving, it has to be the case that any reduction step (evaluation or non-evaluation) preserves the record's behavior.

Since there is more than one way to evaluate a record by evaluating its different components, we need to show that we get the same record at the end. This property is called confluence of evaluation and is proven in section 3.3.

It would have been desirable to show the same property for the general reduction step $\rightarrow$: if there are two different ways to reduce a record, then there is always a way to reduce the two results to the same record. This property is called confluence. However, confluence of $\rightarrow$ does not hold in our calculus, as shown in section 3.2. Instead we were able to show two more limited properties that deal with interactions between a non-evaluation and an evaluation steps. These properties, which we call Basic Strong Lift and Basic Strong Project, and their relation to meaning preservation are discussed in section 3.4.

3.2 Non Confluence of Reduction in the Calculus of Records

Confluence of a reduction means that if this reduction can reduce a record in two different ways, then these two results can be reduced so that they end up the same.

For example:

$$[f \mapsto g + 2, g \mapsto 3 + 5]$$

This can be evaluated in two ways. The first one yields:

$$[f \mapsto 3 + 5 + 2, g \mapsto 3 + 5]$$

The second one yields

$$[f \mapsto g + 2, g \mapsto 8]$$

Now it is fairly intuitive how these solutions can be brought to an equal solution. The two new records will eventually be $[f \mapsto 13, g \mapsto 8]$. For the purposes of being precise, I will show how this happens.

The first record is reduced like this:

$$\begin{array}{l}[f \mapsto 3 + 5 + 2, g \mapsto 3 + 5] \rightarrow \\ [f \mapsto 10, g \mapsto 3 + 3] \rightarrow \\ [f \mapsto 10, g \mapsto 8]\end{array}$$

The second one is reduced to the same record:

$$\begin{array}{l}[f \mapsto g + 2, g \mapsto 8] \rightarrow \\ [f \mapsto 8 + 2, g \mapsto 8] \rightarrow \\ [f \mapsto 10, g \mapsto 8]\end{array}$$

These two records can be reduced to the same solution. Based on this example, one might think that reduction $\rightarrow$ is confluent in our calculus. However, confluence does not always hold. Sometimes records can be reduced in two different ways, and these reductions will be such that they can never be made equal to each other. For example, consider the record:

$$[f(x) \mapsto g, g(y) \mapsto f]$$

This record can be reduced in two different ways. The first one yields:

$$[f(x) \mapsto f, g(y) \mapsto f]$$

The second yields:

$$[f(x) \mapsto g, g(y) \mapsto g]$$

There is no way to bring these two records to a new, equal state. If you can reduce a record in two different ways such that the two results can never be brought together, then the reduction $\rightarrow$ is said to be non-confluent.

3.3 Confluence of Evaluation

Even though the general calculus reduction $\rightarrow$ is not confluent, its subset evaluation $\Rightarrow$ is confluent. This means that if a record is evaluated in two different ways, that the resulting records can eventually be evaluated to the same record.

For example, given a record $[y \mapsto x + 5, x \mapsto 3 + 3]$, the final result of evaluation is $[y \mapsto 11, x \mapsto 6]$ no matter in what order we evaluate the two records. We could substitute $3 + 3$ into the first component first, or we can evaluate x first and substitute 6 in after that. Confluence of evaluation is an essential property for meaning preservation because it guarantees that the evaluation step itself cannot change the meaning of a record (recall that the meaning is defined in terms of the final result of evaluation, see section 3.1).

Here is a more extended example. The record is evaluated two different ways, but both evaluations end up as the same record in the end.

$$
\begin{array}{ll}
[m \mapsto n+2, n \mapsto 3+5] & \Rightarrow \\
[m \mapsto 3+5+2, n \mapsto 3+5] & \Rightarrow \\
[m \mapsto 10, n \mapsto 3+5] & \Rightarrow \\
[m \mapsto 10, n \mapsto 8] & \\
[m \mapsto n+2, n \mapsto 3+5] & \Rightarrow \\
[m \mapsto n+2, n \mapsto 8] & \Rightarrow \\
[m \mapsto 8+2, n \mapsto 8] & \Rightarrow \\
[m \mapsto 10, n \mapsto 8] & \Rightarrow
\end{array}
$$

The proof of this property involves considering all possible cases of the two evaluation steps, and is not given here. See [2] for details.

3.4 Basic Strong Lift and Basic Strong Project

Our goal is to show that a transformation step does not change the meaning, or "behavior," of a record. Since evaluation steps are confluent, we only need to show that non-evaluation steps do not change the meaning of a record. Since the meaning is defined in terms of the result of evaluation, as a minimum we need to show that non-evaluation steps preserve evaluation steps and that after taking an evaluation step and a non-evaluation step from the same record we can bring the results back together by some sequence of reduction steps.

The two properties that deal with such interactions between evaluation and non-evaluations are *basic strong lift* and *basic strong project* defined below. These properties involve a single evaluation step and a single non-evaluation steps. The extensions of these properties to a sequence of evaluation steps are called *strong lift* and *strong project*, respectively, and were introduced in [5]. Historically strong lift and strong project are, in turn, generalizations of the properties *lift* and *project*, first introduced in [4] and discussed in detail in [3]. Strong lift, strong project, and confluence of evaluation together imply meaning preservation of the calculus. The basic strong lift and the basic strong project can be used as base cases for the strong lift and the strong project properties.

Let us define $\leftrightarrow$ as a sequence of forward and backward reduction step, i.e. $D \leftrightarrow D'$ if and only if there are records $D'_1, D'_2, \cdots D'_n$, $n \geq 2$, where $D'_1 = D$, $D'_n = D'$, and $D'_i \rightarrow D'_{i+1}$ or $D'_{i+1} \rightarrow D'_i$ for all i.

Basic strong project. Given $D_1 \Rightarrow D_2$ and $D_1 \hookrightarrow D_3$, there exists D_4 such that $D_3 \Rightarrow D_4$ and $D_2 \leftrightarrow D_4$.

Basic strong lift. Given $D_1 \Rightarrow D_2$ and $D_2 \hookrightarrow D_3$, there exists D_4 such that $D_1 \Rightarrow D_4$ and $D_4 \leftrightarrow D_3$.

Here are a few examples that illustrate basic lift and basic project. Note that they can serve as examples of both properties, depending on which of the two evaluation steps is considered to be the given one, and which one is the constructed one. For instance, in the first example below if we assume that we are given the steps $[n \mapsto m + (2+3), m \mapsto 1] \hookrightarrow [n \mapsto m+5, m \mapsto 1]$ and $[n \mapsto m + (2+3), m \mapsto 1] \Rightarrow [n \mapsto 1 + (2+3), m \mapsto 1]$, then the rest of the reductions complete the example of the basic strong project property, where $D_4 = [n \mapsto 1+5, m \mapsto 1]$. On the other hand, if you are given $[n \mapsto m + (2+3), m \mapsto 1] \hookrightarrow [n \mapsto m+5, m \mapsto 1] \Rightarrow [n \mapsto 1+5, m \mapsto 1]$, then the remaining steps complete the basic strong lift example, with $D_4 = [n \mapsto 1 + (2+3), m \mapsto 1]$.

$$
\begin{array}{ll}
[n \mapsto m + (2 + 3), m \mapsto 1] & \hookrightarrow \\
[n \mapsto m + 5, m \mapsto 1] & \Rightarrow \\
[n \mapsto 1 + 5, m \mapsto 1] & \\
[n \mapsto m + (2 + 3), m \mapsto 1] & \Rightarrow \\
[n \mapsto 1 + (2 + 3), m \mapsto 1] & \hookrightarrow \\
[n \mapsto 1 + 5, m \mapsto 1] &
\end{array}
$$

Below is an example with a black hole:

$$
\begin{array}{ll}
[n \mapsto n + (2 + 3)] & \hookrightarrow \\
[n \mapsto n + 5] & \Rightarrow \\
[n \mapsto \bullet] & \\
[n \mapsto n + (2 + 3)] \Rightarrow & \\
[n \mapsto \bullet] &
\end{array}
$$

This example illustrates a function. Note that the resulting steps involve a duplicated evaluation step since 2 + 3 was copied unevaluated.

$$
\begin{array}{ll}
[f(x) \mapsto x + x, n \mapsto f(2 + 3)] & \hookrightarrow \\
[f(x) \mapsto x + x, n \mapsto f(5)] & \Rightarrow \\
[f(x) \mapsto x + x, n \mapsto 5 + 5] & \\
[f(x) \mapsto x + x, n \mapsto f(2 + 3)] & \Rightarrow \\
[f(x) \mapsto x + x, n \mapsto (2 + 3) + (2 + 3)] & \Rightarrow\Rightarrow \\
[f(x) \mapsto x + x, n \mapsto 5 + 5] &
\end{array}
$$

The proof of these properties involves considering all possible combinations for evaluation and non-evaluation steps, and is not given here. See [2] for details.

4 Summary of Results and Future Work

We have developed a call-by-name calculus of records and have proven its properties listed below:

- Confluence of evaluation
- Basic strong lift
- Basic strong project

Our goal is to prove that any transformation of this calculus that involves any sequence of forward and backward $\rightarrow$ step preserves the meaning of a record, as defined by the evaluation $\Rightarrow$. The relationship between the three properties proven in this calculus and the meaning preservation property is explained in detail in section no 3.4.

The future work on this paper will be to take the three properties that we have proven and then combine them inductively for a proof of strong lift and project which in turn implies the meaning preservation.

References

[1] Z. M. Ariola and J.W. Klop.*Cyclic Lambda Graph Rewriting.* In *Proc. of the Eight IEEE Symposium on Logic in Computer Science*, Paris, July 1994.

[2] Emily Christiansen, Elena Machkasova *The call-by-name calculus of records and its basic properties.* Technical Report, UMM, 2006.

[3] Elena Machkasova *Computational Soundness of Non-Confluent Calculi with Applications to Modules and Linking*, PhD Dissertation, April 2002, Boston University.

[4] Elena Machkasova and Franklyn A. Turbak *A calculus for link-time compilation.* In *Programming Languages and Systems*, 9th European Symp. Programming, volume 1782 of LNCS, pages 260-274 Springer-Verlag, 2000

[5] J. B. Wells, Detlef Plump, and Fairouz Kamareddine.*Diagrams for meaning preservation.* In *Rewriting Techniques and Applications, 14th Int'l Conf., RTA 2003,* volume 2706 of *LNCS*, pages 88-106. Springer-Verlag, 2003.