Isomorphisms between Two Groupoids : An Experiment in Program Synthesis and Transformation

F Esfandiari
Dept. of Computer Science
Queen Mary and Westfield College
University of London
fard@dcs.qmw.ac.uk

September 21, 1994

Abstract

The research work which comes under the heading 'program transformation' started in the seventies. The term coveres both the conversion of specifications to runnable programs (though the term 'program synthesis' is sometimes preferred here) and also the conversion of existing programs into equivalent and more efficient ones. Our main concern in this paper is on transforming programs, in particular finding efficient programs. The problem of finding the isomorphisms between two finite groupoids has been considered. Initially, by using unfold/fold method, a program for this problem is derived from its definition which then leads to the efficient program by using promotion (fusion), unfold/fold method and necessary proved lemmas. We show that during the process of both synthesis and transformation there are some stages for which we need some lemmas to proceed on. Then generalizations of these lemmas are proved.

1 Introduction

The research work which comes under the heading 'program transformation' started in the seventies. The term coveres both the conversion of specifications to runnable programs (though the term 'program synthesis' is sometimes preferred here) and also the conversion of existing programs into equivalent and more efficient ones. Our main concern in this paper is on transforming programs, in particular finding efficient programs. Many different tactics have been suggested for program transformation. Three of them should be mentioned here, as unfold/fold[BrD77], accumulation[B84], and [B88] and promotion [W88], [C90] and [B88]. In the first section a problem will be defined and a program will be derived from its definition. In the second section the derived program is transformed to an efficient one via a series of steps. Finally, in the last section we talk about the results we have got in this paper plus future works.
In the following sections we use some conventions. Capital letters will be used for variables which are bound by logical quantifiers and by set comprehensions. The notations $\wedge, \vee, \Rightarrow$, and $\Leftrightarrow$ represent the logical connections, $fst(a,b) = a$ and $snd(a,b) = b$. The notation '$\gg$' will be used to show every simplification step. The notation '$\in$' stands for the function '*member*' (the following equations).

$a \in \emptyset = false$
$a \in (\{b\} \cup x) = (a = b) \vee a \in x$

Functions *map* and *filter* will be used with the following equations.

$map\ f\ \emptyset = \emptyset$
$map\ f\ (\{a\} \cup x) = \{f\ a\} \cup map\ f\ x$

$filter\ p\ \emptyset = \emptyset$
$filter\ p\ (\{a\} \cup x) = (if\ (p\ a)\ then\ \{a\}\ else\ \emptyset) \cup filter\ p\ x$

Also, we use the following three functions *mset* (make set), *uEO* (union each on) and *flatten* in the forthcoming sections repeatedly.

$mset\ a = \{a\}$

$uEO\ xs\ y = map\ (y\cup)\ xs$

$flatten\ \emptyset = \emptyset$
$flatten\ (\{a\} \cup x) = a \cup flatten\ x$

It is suitable to mention that *flatten* above is big union ($\bigcup$).

2 Synthesis of the Problem

By definition isomorphisms between two groupoids are their homomorphic functions which keep the one to one and onto properties. Functions themselves belong to the set of relations between two sets which are power set of cartesian product of those two groupoids. So the first task is to get a function (*cp*) to calculate cartesian product of two sets. This function can be defined by:

$cp(s_1, s_2) = \{(A, B) \mid A \in s_1 \wedge B \in s_2\}$

Using the unfold/fold method we may end up with the following definition for *cp* [D75]:

$$\left.\begin{array}{l} cp(\emptyset, s_2) = \emptyset \\ cp(\{a\} \cup x, s_2) = map\ (pair\ a)\ s_2 \cup cp(x, s_2) \end{array}\right\} \quad [1]$$

where $pair\ a\ b = (a, b)$.

To get the definition of power set of two sets, we need the definition of subset of two sets which is given below.

$x \subset y = \forall A\ .\ A \in x \rightarrow A \in y$

By using unfold/fold method, we can get the following equations for $\subset$.

$\emptyset \subset y = true$
$(\{a\} \cup x) \subset y = a \in y \wedge (x \subset y)$

Now, the definition of *ps*, i.e. the function that is to find power set of two sets, can be given as:

$ps(x) = \{Z \mid Z \subset x\}$

We can apply the method of unfold/fold to the above definition to get the required equations for *ps*. The result is a set of equations which is not terminating. This non-termination happens when, in the definition of *ps*, we substitute x with a non-empty set:

$ps(\{a\} \cup w) = \{Z \mid Z \subset (\{a\} \cup w)\}$

Since $\subset$ has been defined by recursion on its first argument, we cannot continue any furthur. One method to solve this problem, is using a technique which is called *narrowing*([Re85]). Below, we use narrowing for sets, that is to say for any set, for example Z, there are two cases to be considered: Z is either empty set or there exists an element B and a set Y such that $Z = \{B\} \cup Y$. Then:

$\{Z \mid Z \subset (\{a\} \cup w)\}$
$\{\emptyset \mid \emptyset \subset x\} \cup \{\{B\} \cup Y \mid (\{B\} \cup Y) \subset (\{a\} \cup w)\} \gg$
$\{\emptyset \mid true\} \cup \{\{B\} \cup Y \mid (\{B\} \cup Y) \subset (\{a\} \cup w)\} \gg$
$\{\emptyset\} \cup \{\{B\} \cup Y \mid (\{B\} \cup Y) \subset (\{a\} \cup w)\} \gg$
$\{\emptyset\} \cup \{\{B\} \cup Y \mid B \in (\{a\} \cup w) \wedge Y \subset (\{a\} \cup w)\} \gg$
$\{\emptyset\} \cup \{\{B\} \cup Y \mid (B = a \vee B \in w) \wedge Y \subset (\{a\} \cup w)\} \gg$
$\{\emptyset\} \cup \{\{B\} \cup Y \mid (B = a \wedge Y \subset (\{a\} \cup w)) \vee (B \in w \wedge Y \subset (\{a\} \cup w))\} \gg$
$\{\emptyset\} \cup \{\{B\} \cup Y \mid B = a \wedge Y \subset (\{a\} \cup w)\} \cup \{\{B\} \cup Y \mid B \in w \wedge Y \subset (\{a\} \cup w)\} \gg$
$\{\emptyset\} \cup \{\{a\} \cup Y \mid Y \subset (\{a\} \cup w)\} \cup \{\{B\} \cup Y \mid B \in w \wedge Y \subset (\{a\} \cup w)\}$

Since '$Y \subset (\{a\} \cup w)$' appears in each set comprehension expression (up to name of the first argument of $\subset$), then this process does not reach to a terminating program. To solve this problem, we will prove the following lemma for the function $\subset$ that can be proved by considering different cases on the left and right hand side of $\subset$.

Lemma 2.1 *For the function* $\subset$*, the following equations hold.*

$x \subset \emptyset = (x = \emptyset)$
$x \subset (\{a\} \cup w) = (\exists T\ .\ x = (\{a\} \cup T) \wedge T \subset w) \vee x \subset w$

With the above lemma, we can get the required equations for ps. Let us see how is the development of the definition of ps this time.

$ps\ \emptyset =$
$\quad \{Z \mid Z \subset \emptyset\} \gg$
$\quad \{Z \mid Z = \emptyset\} \gg$
$\quad \{\emptyset\}$
$ps\ (\{a\} \cup w) =$
$\quad \{Z \mid Z \subset (\{a\} \cup w)\} \gg$
$\quad \{Z \mid (\exists T\ .\ Z = (\{a\} \cup T) \wedge T \subset w) \vee Z \subset w\} \gg$
$\quad \{Z \mid (\exists T\ .\ Z = (\{a\} \cup T) \wedge T \subset w)\} \cup \{Z \mid Z \subset w\} \gg$
$\quad \{\{a\} \cup T \mid T \subset w\} \cup \{Z \mid Z \subset w\} \gg$
$\quad \{\{a\} \cup T \mid T \in (ps\ w)\} \cup \{Z \mid Z \subset w\} \gg$
$\quad \{\{a\} \cup T \mid T \in (ps\ w)\} \cup ps\ w \gg$
$\quad (f\ a\ (ps\ w)) \cup ps\ w$

where f is defined by $f\ a\ u = \{\{a\} \cup T \mid T \in u\}$. By instantiating u in the definition of f, we can get a recursive program for f. Since this recursive program for '$f\ a\ u$' happens to be equivalent to '$map\ (\{a\}\cup)\ u$', to avoid giving unnecessary functions, we will use this equivalent form. That is, the following program for ps will be used.

$$\left.\begin{array}{l} ps\ \emptyset = \{\emptyset\} \\ ps\ (\{a\} \cup w) = (map\ (\{a\}\cup)\ (ps\ w))\ \cup\ ps\ w \end{array}\right\} \quad [2]$$

As stated before, the set of relations between two sets is power set of cartesian product of those sets. Let rs be the function that gets this set of relations. Then:

$$rs\ s_1\ s_2 = ps\ (cp\ (s_1, s_2)) \quad [3]$$

Having got the set of relations between two sets, the next task is to get the test functions which will check the functionality, injective, surjective and homomorphism properties. The definitions of the first three test functions are given below in which each function gets a relation as input (r below) and gives $false$ or $true$ as output.

$is\text{-}fun\ r =$
$\quad \forall A, B.\ A, B \in r \Rightarrow (fst\ A \neq fst\ B) \vee (fst\ A = fst\ B \wedge snd\ A = snd\ B)$
$is\text{-}inj\ r =$
$\quad \forall A, B.\ A, B \in r \Rightarrow (snd\ A \neq snd\ B) \vee (snd\ A = snd\ B \wedge fst\ A = fst\ B)$
$is\text{-}surj\ y\ r =$
$\quad \forall A.A \in y \Rightarrow \exists B.B \in r \wedge snd\ B = A$

With unfold/fold method the following recursive programs can be derived for $is\text{-}fun$, $is\text{-}inj$ and $is\text{-}surj$ from the above specification.

$is\text{-}fun\ r = is\text{-}fun_1\ r\ r$
$is\text{-}fun_1\ \emptyset\ r\ = true$
$is\text{-}fun_1\ (\{p\} \cup w)\ r = is\text{-}fun_2\ p\ r \wedge is\text{-}fun_1\ w\ r$
$is\text{-}fun_2\ p\ \emptyset\ = true$
$is\text{-}fun_2\ p\ (\{p'\} \cup w)\ = (fst\ p \neq fst\ p' \vee (fst\ p = fst\ p' \wedge snd\ p = snd\ p')) \wedge is\text{-}fun_2\ p\ w$

$is\text{-}inj\ r = is\text{-}inj_1\ r\ r$
$is\text{-}inj_1\ \emptyset\ r = true$
$is\text{-}inj_1\ (\{p\} \cup w)\ r = is\text{-}inj_2\ p\ r\ \wedge is\text{-}inj_1\ w\ r$
$is\text{-}inj_2\ p\ \emptyset = true$
$is\text{-}inj_2\ p\ (\{p'\} \cup w) = (snd\ p \neq snd\ p' \vee (snd\ p = snd\ p' \wedge fst\ p = fst\ p')) \wedge is\text{-}inj_2\ p\ w$

$is\text{-}surj\ \emptyset\ r = true$
$is\text{-}surj\ \ (\{b\} \cup w)\ r = is\text{-}surj_1\ r\ b\ \wedge is\text{-}surj\ w\ r$
$is\text{-}surj_1\ \emptyset\ b\ = false$
$is\text{-}surj_1\ (\{p\} \cup w)\ b\ = (snd\ p = b) \vee is\text{-}surj_1\ w\ b$

Therefore we can get the bijective functions that exist between two given finite sets. Now, let us consider two finite groupoids, that is two finite sets plus one operator for each groupoid. To find the isomorphisms between two finite groupoids, we have to have a function that checks the

homomorphism property: a function whose input is a relation and its output is $false$ or $true$. Let r be a relation between two given finite groupoids and let g_1 and g_2 be the operators of these groupoids. Then The definition of the function that will check the homomorphism property is given below.

$is\text{-}hom\ r =$

$\forall A, B.\ A, B \in r \Rightarrow \exists C.C \in r \wedge (g_1(fst\ A, fst\ B), g_2(snd\ A), snd\ B)) = C$

Again, with unfold/fold method, we can get the following recursive program for $is\text{-}hom$.

$is\text{-}hom\ r = is\text{-}hom_1\ r\ r\ r$

$is\text{-}hom_1\ \emptyset\ y\ z = true$

$is\text{-}hom_1\ (\{p\} \cup w)\ y\ z = is\text{-}hom_2\ p\ y\ z \wedge is\text{-}hom_1\ w\ y\ z$

$is\text{-}hom_2\ p\ \emptyset\ z = true$

$is\text{-}hom_2\ p\ (\{p'\} \cup w)\ z = is\text{-}hom_3\ p\ p'\ z \wedge is\text{-}hom_2\ p\ w\ z$

$is\text{-}hom_3\ p\ p'\ \emptyset = false$

$is\text{-}hom_3\ p\ p'\ (\{q\} \cup w) = (g_1(fst\ p), fst\ p'), g_2(snd\ p, snd\ p')) = q \vee is\text{-}hom_3\ p\ p'\ w$

The last task is to collect tested homomorphisms between two groupoids. Let s_1 and g_1 be the set and the operator of a groupoid and let s_2 and g_2 be the set and the operator of another groupoid. Assume that $isomorphisms$ is the function to collect the tested homomorphisms between two groupoids. Then the definition of this function will be:

$isomorphisms\ s_1\ s_2 =$

$\{A \mid A \in (rs\ s_1\ s_2)\ \wedge is\text{-}fun\ A \wedge is\text{-}inj\ A \wedge is\text{-}surj\ s_2\ A \wedge is\text{-}hom\ A\}$

The above definition for isomorphisms has a pattern similar to the definition of cp. Then the definition of $isomorphisms$ can similarly be developed. The result is as follows.

$isomorphisms\ s_1\ s_2 = filter\ (f\ s_2)\ (rs\ s_1\ s_2)$

$where\ \ f\ s_2\ r = is\text{-}fun\ r \wedge is\text{-}inj\ r \wedge is\text{-}surj\ s_2\ r \wedge is\text{-}hom\ r$ [4]

Therefore the process of synthesising of the problem has been completed. In the next section we are going to have some transformation steps which improve the programs we have already got.

3 The Transformation Steps

We got recursive programs for the functions cp, ps, rs and others in the last section. Regarding efficiency, the number of reductions and cpu times used for implementation of these programs for a sample groupoid are shown in Table 4. In this section, we embark on step by step improvement of these programs. First, to improve rs two functions ps and cp can be combined to avoid building unnecessary structures. This method is called 'deforestation' and 'fusion' by Wadler[W88] and Chin[C90] respectively, though Chin's work is not the same as Wadler. Both of them give some strategies to remove unnecessary construction of certain structures in an automatic fashion.
However, we are going to apply another approach for transforming rs. The reason is that we can use the new version of rs for future transformations that we will make. As we can see in [3], based on two variables s_1 and s_2, the result of cp will be passed to ps. In [1], where s_1 is a non-empty set, the result of cp (right hand side) is union of two parts. Then this union, will be passed to ps. Therefore, in [3], the input of ps may be the union of two sets rather than union of a singleton set and another set (look at the definition of ps in [2]). This will rise a question that if input of ps is union of two sets, then what is the result. The following lemma has the answer to this question.

Lemma 3.1 *For the function ps we have:*

$ps(u \cup v) = flatten(map\ (uEO\ (ps\ u))\ (ps\ v))$

This leads to a fusion for the composition of ps and cp. That is;

$rs\ \emptyset\ y\ =$

$\quad ps(cp(\emptyset, y)) \gg$

$\quad ps(\emptyset) \gg$

$\quad \{\emptyset\}$

$rs\ (\{a\} \cup x)\ y\ =$

$\quad ps(cp(\{a\} \cup x, y)) \gg$

$\quad ps(cp(x, y) \cup map\ (pair\ a)\ y) \gg$

$flatten(map\ (uEO\ ps(cp(x,y)))\ ps(map\ (pair\ a)\ y)) \gg$
$flatten(map\ (uEO\ (rs\ x\ y))\ ps(map\ (pair\ a)\ y))$

Therefore, from now on, we can use the following equation for rs.

$rs\ \emptyset\ y\ = \{\emptyset\}$

$rs\ (\{a\} \cup x)\ y\ = flatten(map\ (uEO\ (rs\ x\ y))\ ps(map\ (pair\ a)\ y))$

Having got this improvement for rs, now we consider the equation [4]. From this equation we understand that the result of the function rs, the set of relations between two sets, will be filtered by four test functions: *is-fun*, *is-inj*, *is-surj* and *is-hom*. Then again it is useful to avoid the construction of unnecessary structures: fusion idea. To do this, some equations, that will be used in the fusion process, should be proved. These fusions have specific characteristics that make them special in the sense that they can not be done automatically by the algorithms which have been suggested in[W88] and in[C90]. This is because of the nature of the test functions, i.e. *is-fun*, *is-inj*, *is-surj* and *is-hom*, which we have in our program. These test functions have such properties that allow us to delete some parts of the constructions of the program for being completely unnecessary. Let us show a couple of simple cases. Let p be a predicate and xs be a set. Consider an equation of a function where '$filter\ p\ xs$' is on the right hand side and suppose that we know $p\ x = false$ for all $x \in xs$. Then we can use the equation '$filter\ p\ xs = \emptyset$' in our track of transformation. As another example, consider two pieces of '$if\ p(xs \cup ys)\ then\ exp_1\ else\ exp_2$' and '$if\ (p\ xs \wedge p\ ys)\ then\ exp_1\ else\ exp_2$' and we need to transform one of them to the other. This is possible if '$p(xs \cup ys)$' and '$(p\ xs \wedge p\ ys)$' are equivalent. Several of such patterns in transforming the functions are given in this section. We use these opportunities to prove some lemmas that hold for other test functions similar to those above.

Let us start with the first fusion in which we try to make the function $funs$ from the composition of rs with *is-fun*. The natural way to do this is to give the following definition and to try to transform it through instantiating x.

$funs\ x\ y = filter\ is\text{-}fun\ (rs\ x\ y)$.

The problem shows itself when we are going to get the result for the case where x is not empty set. Let us have $x = \{a\} \cup z$. Then:

$funs\ (\{a\} \cup z)\ y =$
$filter\ is\text{-}fun\ (rs\ (\{a\} \cup z)\ y) \gg$
$filter\ is\text{-}fun\ \ flatten(map\ (uEO\ (rs\ z\ y))\ ps(map\ (pair\ a)\ y)) \gg$
$flatten(map\ g\ ps(map\ (pair\ a)\ y))$
$where\ g\ x = filter\ is\text{-}fun\ ((uEO\ (rs\ z\ y))\ x)$

In '*where*' part of the last line, to have folding with the definition of $funs$ we need to place the test function '*is-fun*' exactly before '$rs\ z\ y$'. This is not possible in an automatic fashion. However, we can continue this track of transforming the function $funs$ through some equations that proofs of these equations depend on statements of the following lemma. Below in the next lemma and afterwards, the notation '$-$' stands for difference operator between two sets and '$\circ$' for composition of two functions.

Lemma 3.2 *Let xs and ys be sets, p be a predicate and let g be a function from a type α to α. Then:*

1. $(\forall x.\ x \in xs \Rightarrow p\ x = true) \Leftrightarrow (filter\ p\ xs = xs)$.
2. $(\forall x.\ x \in xs \Rightarrow p\ x = false) \Leftrightarrow (filter\ p\ xs = \emptyset)$.
3. $(\forall x.\ x \in xs \Rightarrow p(g\ x) = true) \Leftrightarrow (filter\ p\ (map\ g\ xs) = map\ g\ xs)$.
4. $(\forall x.\ x \in xs \Rightarrow p(g\ x) = false) \Leftrightarrow (filter\ p\ (map\ g\ xs) = \emptyset)$.
5. $(\forall x, u.\ x \in xs \wedge u \in ps(map\ h\ (xs - \{x\})) \Rightarrow p(g(\{h\ x\} \cup u)) = false) \Leftrightarrow$
   $filter\ p\ (map\ g\ (ps(map\ h\ xs))) = filter\ p\ (\{g\ \emptyset\})$.
6. $(\forall x, y, u.\ x, y \in xs \wedge u\ is\ any\ set \Rightarrow p(\{h\ x, h\ y\} \cup u) = false) \Leftrightarrow$
   $filter\ p\ ps(map\ h\ xs) = filter\ p\ (\{\emptyset\} \cup (map\ (mset \circ h)\ xs))$.

7. $(\forall x.\ x \in xs \Rightarrow p(g\ x) = p\ x) \Leftrightarrow map\ g\ (filter\ p\ xs) = filter\ p\ (map\ g\ xs)$

8. $\forall x, z.\ x \in xs \Rightarrow p\ (x \cup z) = p\ x \wedge p\ z \Leftrightarrow$
$filter\ p\ (uEO\ xs\ z) = if\ (p\ z)\ then\ flatten(uEO\ (filter\ p\ xs)z)\ else\emptyset$

9. $\forall x, y.\ x \in xs \wedge y \in ys \Rightarrow p\ (x \cup y) = p\ x \wedge p\ y \Leftrightarrow$
$filter\ p\ flatten(map\ (uEO\ xs)\ ys) = flatten(map\ (uEO\ (filter\ p\ xs))\ (filter\ p\ ys))$

Now, with the help of the above lemma, let us get a couple of equations whose existence are necessary to continue the transformation of *funs*.
For given sets xs and ys, since '$is\text{-}fun\ (\{(pair\ a)\ x, (pair\ a)\ y\} \cup u) = false$' for any $x, y \in xs$, then by Lemma 3.2(6) we can use the following equation.

$filter\ is\text{-}fun\ (ps(map\ (pair\ a)\ xs)) =$
$filter\ is\text{-}fun\ (\{\emptyset\} \cup (map\ (mset \circ (pair\ a))\ xs)) \gg$
$(filter\ is\text{-}fun\ \{\emptyset\}) \cup (filter\ is\text{-}fun\ (map\ (mset \circ (pair\ a))\ xs)) \gg$
$\{\emptyset\} \cup (filter\ is\text{-}fun\ (map\ (mset \circ (pair\ a))\ xs)) \gg$
$\{\emptyset\} \cup (map\ (mset \circ (pair\ a))\ xs)$ 3.2(3)

For the last line of the above simplification, we have used Lemma 3.2(3) because '$is\text{-}fun\ (mset \circ (pair\ a))\ x = true$' for all $x \in xs$. That is we have:

$filter\ is\text{-}fun\ (ps(map\ (pair\ a)\ xs)) = \{\emptyset\} \cup (map\ (mset \circ (pair\ a))\ xs)$

Suppose $a \notin z$. Then, for any $u \in (rs\ z\ y)$ and any $v \in ps(map\ (pair\ a)\ y)$, we should have '$is\text{-}fun(u \cup v) = is\text{-}fun\ u \wedge is\text{-}fun\ v$'. This fact makes it possible to use Lemma 3.2(9) and so having the following equation.

$filter\ is\text{-}fun\ flatten(map\ (uEO\ (rs\ z\ y))\ ps(map\ (pair\ a)\ y)) =$
$flatten(map\ (uEO\ (filter\ is\text{-}fun\ (rs\ z\ y)))\ (filter\ is\text{-}fun\ ps(map\ (pair\ a)\ y)))$

Therefore by using the above rules and unfold/fold method we get the following equations for *funs*.

$funs\ \emptyset\ y = \{\emptyset\}$
$funs\ (\{a\} \cup x)\ y = (funs\ x\ y) \cup flatten(map\ (f\ a\ x\ y)\ y)$
$\quad where\ f\ a\ x\ y\ b = map\ (\{(a, b)\}\cup)\ (funs\ x\ y)$

After finding the equations of *funs*, somehow, we have to delete the test function *is-inj*. Let *injections* be defined as follows.

$injections\ x\ y = filter\ is\text{-}inj\ (funs\ x\ y)$

Not surprisingly, the process of transforming the definition of *injections* to its new equations is not without trouble: we need the following lemma.

Lemma 3.3 *Let a and b be two elements and z and y be two sets. Then:*
$filter\ is\text{-}inj\ (map\ (\{(a, b)\}\cup)\ (funs\ z\ y)) =$
$\quad map\ (\{(a, b)\}\cup)\ (filter\ is\text{-}inj\ (funs\ z\ (y\text{-}\{b\})))$

With the help of the above lemma and by unfold/fold method, the following equations can be derived for *injections*.

$injections\ \emptyset\ y = \{\emptyset\}$
$injections\ (\{a\} \cup x)\ y = injections\ x\ y \cup flatten(map\ (f\ a\ x\ y)\ y)$
$\quad where\ f\ a\ x\ y\ b = (map\ (\{(a, b)\}\cup)\ (injections\ x\ (y\text{-}\{b\})))$

To get the bijections of the two sets, the function *is-surj* should be composed with *injection*. Let *bijections* be defined by:

$bijections\ x\ y = filter\ (is\text{-}surj\ y)\ (injections\ x\ y)$.

Again, the process of transforming the definition of *bijections* to its new equations is not without trouble. By having the following lemma, it is possible to get new equations for *bijections*.

Lemma 3.4 *Let a and b be two elements and z and y be two sets. Then:*
$filter\ (is\text{-}surj\ y)\ (map\ (\{(a, b)\}\cup)\ (injections\ z\ (y\text{-}\{b\}))) =$
$map\ (\{(a, b)\}\cup)\ (filter\ (is\text{-}surj\ (y\text{-}\{b\}))\ (injections\ z\ (y\text{-}\{b\})))$

Then the following equations can be derived for *bijections*.

$bijections\ \emptyset\ \emptyset = \{\emptyset\}$
$bijections\ \emptyset\ (\{c\} \cup w) = \emptyset$
$bijections\ (\{a\} \cup x)\ y = bijections\ x\ y \cup flatten(map\ (f\ a\ x\ y)\ y)$
$\quad where\ f\ a\ x\ y\ b = map\ (\{(a,b)\}\cup)\ (bijections\ x\ (y\text{–}\{b\}))$

Finally, to get the isomorphisms between two groupoids, as usual, we have to get new equations for the function *iso* which is defined by:

$iso\ s_1\ s_2 = filter\ is\text{-}hom\ (bijections\ s_1\ s_2)$

The following lemma is useful for the process of finding new equations for *iso*.

Lemma 3.5 *Let s_1 and s_2 be two sets that, respectively with a operator, make two different groupoids. Let $s_1 = \{a\} \cup x$. Then:*

$filter\ is\text{-}hom\ (bijections\ x\ s_2) = \emptyset$

The above lemma helps to derive the following equations for *iso*.

$$\left.\begin{array}{l} iso\ \emptyset\ \emptyset = \{\emptyset\} \\ iso\ \emptyset\ (\{c\} \cup w) = \emptyset \\ iso\ (\{a\} \cup x)\ y = flatten(map\ (f\ a\ x\ y)\ y) \\ \quad where\ f\ a\ x\ y\ b = map\ (\{(a,b)\}\cup)\ (filter\ (h\ (a,b))\ bijections\ x\ (y\text{–}\{b\})) \\ \quad where\ h\ p\ r = is\text{-}hom\ (\{p\} \cup r) \end{array}\right\} \quad [5]$$

Here, the track of our transformation has reached to its final stage. In the next section, we would like to have a comparison of the implementation results for the found programs.

4 Conclusion

We started with the definition of the problem of finding the isomorphisms between two finite groupoids. Then a program was derived with less efficiency from that definition. Thereafter by proving some necessary lemmas (in four steps) the original program was transformed to a more efficient program. The methods that are used, for both synthesis and transformation steps are not new, but, can not be done automatically through some techniques which are already well known. This lack of automatism comes from the specific characters that the test functions *is-fun*, *is-inj*, *is-surj* and *is-hom* possess. For example, consider Lemma 3.5. The nature of this lemma is in such a way that we cannot get this lemma through automatic program transformation. This means that for any other program, with such properties and structure similar to the isomorphism program which was discussed here, it is completely useful to use the lemmas, or possibly the extension of them, that have been proved in this paper.

	e	a	b
e	e	a	b
a	a	b	e
b	b	e	a

Table 1: Group Z_3

Here we would like to come to a conclusion by comparing the implementation results of an example. Consider the group Z_3 whose elements and the actions of its operator are given in Table 4.
There are two isomorphisms between Z_3 and itself. We have tried to get this result through implementation of several programs: *isomorphisms*([4]) is the original program that is derived from its specification, *iso*([5]) is found after four steps of transformation and iso_1, iso_2 and iso_3 whose definitions are as follows.

$iso_1\ s_1\ s_2 = filter\ (f\ s_2)\ (funs\ s_1\ s_2)$
$\quad where\ f\ s_2\ r = is\text{-}inj\ r \wedge is\text{-}surj\ s_2\ r \wedge is\text{-}hom\ r$
$iso_2\ s_1\ s_2 = filter\ (f\ s_2)\ (injections\ s_1\ s_2)$
$\quad where\ f\ s_2\ r = is\text{-}surj\ s_2\ r \wedge is\text{-}hom\ r$
$iso_3\ s_1\ s_2 = filter\ (f\ s_2)\ (bijections\ s_1\ s_2)$

	reductions	cells claimed	cpu time
isomorphisms	177649	203345	3.62
iso_1	29065	32251	0.62
iso_2	15266	17591	0.32
iso_3	11472	13374	0.23
iso	10504	12169	0.22

Table 2: Implementation results for isomorphism programs

$$where \;\; f\ s_2\ r = is\text{-}hom\ r$$

For these implementations, that have been done in Miranda, the number of reductions, cells claimed and the used cpu times for each program have been accounted to get the efficiency of the different programs. The result can be seen in Table 4.

References

[BrD77] Burstall R.M. and Darlington J.: *A Transformation System for Developing Recursive Programs.* Journal of the ACM, **24**, 1, pp. 44-67 (1977).

[B84] Bird R.S.: *The promotion and Accumulation Strategies in Transformational Programming.* ACM Transactions on Programming Languages and Systems, **6**, 4, (1984).

[B88] Bird R.S.: *Lectures on Constructive Functional Programming.* Technical Monograph PRG-69, Oxford University Computing Laboratory, Programming Research Group, Sep. 1988.

[C90] Chin W.N.: *Automatic Methods for Program Transforming.* PhD Dissertation, University of London, Imperial College. (1990).

[D75] Darlington J.: *Application of program transformation to program synthesis.* Proc. IRIA symp. Proving and Improving Programs. Arc-et-Senans. France, pp. 133-144 (1975).

[MW71] Manna Z. and Waldinger R.: *Towards automatic program synthesis.* CACM, **14**, 3, pp. 151-165 (1971).

[Re85] Reddy U.S.: *Narrowing as the Operational Semantics of Functional Language* Symposium on Logic Programming, IEEE, Boston, pp. 138-151 (1985).

[W88] Wadler P.: *Deforestation: Transforming Programs to Eliminate Trees.* European Symposium on Programming, Nancy, France. pp. 344-358 (1988).