J. Functional Programming 3 (3): 283–313, July 1993

Program transformation with metasystem transitions

VALENTIN F. TURCHIN
The City College of New York

Abstract

A new program transformation method is presented. It is a further refinement of supercompilation where the supercompiler is not applied directly to the function to be transformed, but to a metafunction, namely an interpreter which computes this function using its definition and an abstract (i.e. including variables) input. It is shown that with this method such tranformations become possible which the direct application of the supercompiler cannot perform. Examples include the merging of iterative loops, function inversion, and transformation of deterministic into non-deterministic algorithms, and *vice-versa*.

Capsule review

Supercompilation is a program transformation method that can achieve partial evaluation, and is in some respects more powerful.

Turchin presents a method to improve the transformational power of supercompilation without modifying the subject program or the transformation system. The essential idea is to insert an interpretive layer between the subject program and the transformation system. The motivation is to increase the power of the overall program transformation, allowing transformations which the underlying system cannot achieve.

Since the interpreter can take a more abstract view of the computation carried out by the object program and use various rules on the abstract computation histories (associativity, distributivity, unwinding), the introduction of the interpretive layer can lead to improvements outside the scope of ordinary supercompilation.

It is an instance of applying the principle of metasystem transition to program transformation. The potential of the method is shown by giving examples including function inversion and merging iterative loops. Although done by hand, the derivations of the present paper can be automated.

The paper functions as a gentle introduction to supercompilation with specific technical material introduced by need: the language Refal, Refal graphs, walk grammars, metacoding and operations on graphs.

1 Introduction

In this paper we present a program transformation method which is a further development of *supercompilation* as described in Turchin (1986). To express symbolically what is new in this method, we represent as $S_2 \succ S_1$ the relation of S_2 being a metasystem with respect to S_1, which means that S_2 examines, controls, and manipulates

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

S_1. Creation of a metasystem is a *metasystem transition*. We understand functions as machines that manipulate data, so if F is a function, and D its data, the relation F>D holds. The supercompiler (Scp) runs the machine which evaluates a call of function F with unknown values of some variables and constructs a model of this machine by finding a self-sufficient finite set of the machine's configurations and transitions between them. This model becomes a new program. The supercompiler machine is a metasystem with respect to the function machine: Scp>F>D. The construction and use of a supercompiler is a metasystem transition.

Our new method also depends on supercompilation. But instead of just applying Scp to function calls, we make one more metasystem transition: we transform F into the input for a certain interpreter Int, and apply Scp to Int: Scp>Int>F>D. As long as the interpreter is correct, the resulting program defines a function equivalent to F. (We say equivalent, and not just identical, because supercompilation, as we use it, may lead to an extension of the domain of the function; our equivalence is, actually, a partial ordering relation preserving semantics in the minimal domain.)

It is a surprising fact that this, seemingly trivial, operation radically increases the power of the program transformer, allowing such transformations which the supercompiler alone, applied directly to function calls, cannot perform. Moreover, since Int is itself a function, we can again make a metasystem transition and put another Int over it. This may be repeated any number of times creating schemes of the form:

$$\mathtt{Scp} \succ \mathtt{Int} \succ \cdots \succ \mathtt{Int} \succ \mathtt{F} \succ \mathtt{D}$$

It is possible that further metasystem transitions may show even better performance, but this is hard to check manually. (At the moment of this writing the supercompiler we have has not yet been coupled with an interpreter).

The formalism we have developed allows for what in (Turchin, 1977) is called *the stairway effect*: a 'mechanical' way to repeat metasystem transitions. According to the view developed in (Turchin, 1977), the stairway effect is the essence of the phenomenon of evolution; to make mechanical intelligence, we must be able to make repeated metasystem transitions in the machine.

To see why the metasystem transition helps, we must say more about the interpreter Int. Its argument is not just a program for F, but a formal object which can be used as a generator of all possible computation histories of a call of F. Generally, this is a context-free grammar, but in the case of a regular grammar it is more convenient to represent it by a regular expression. Computation histories produced by the grammar or the regular expression are *abstract*, i.e. may include free variables. When we define the interpreter, we endow it with the power to analyze and use the set of all abstract computation histories in various ways. The general principle is that when an input value is given to Int, it picks up from the whole set of abstract computations that computation which will take place with this given input. However, the way Int looks for this specific computation history may vary. It can examine possible computations not only in the order in which the process of computation normally proceeds, i.e. from left to right, but in the opposite direction

(see two kinds of loop unwinding in Section 5). It also may analyze some parts of computation in one direction, while others in the other.

It is this 'clever' interpreter which is supercompiled in order to obtain a finite program from a set, generally infinite, of abstract computation histories.

When the supercompiler is applied directly to `F`, it follows and oversees the work of `F`; the nodes of the graph it constructs are defined by arguments of `F`. In the metasystem transition from `F` to `Int` computation histories of `F` become arguments of `Int`. Now, what the supercompiler analyses and generalizes is computation histories of `F`, not just its arguments. Thus the transformation of the algorithm may be much deeper than it is possible in a direct supercompilation.

In Section 2 we define the language we use, as well as a brief exposition of supercompilation. The language is Refal, and we use it in two forms: Refal programs, which are aimed at the human user, and Refal graphs, which are subject to transformations. In Section 3 we introduce context-free grammars which generate the sets of all possible walks in a Refal graph. Section 4 describes how to do a metasystem transition from a machine to a metamachine. We introduce objects called *metavariables*. Section 5 defines the metafunction `Int` which is placed between the function to be transformed and the supercompiler. In Section 6 we formulate our method of program (graph) transformation. The most significant example, the merging of iterative loops, is presented in Section 7. The transformation is done in three stages, in the course of which the original deterministic algorithm becomes non-deterministic, and then deterministic again. Section 8 gives a simple example of function inversion. The conclusions are made in Section 9.

Speaking of related work, there is a huge literature on program transformation, the closest to our work being the work on partial evaluation (or specialization), which is one part of what the supercompiler is doing. By self-application of a partial evaluator one can automatically obtain compilers and compiler generators (Futamura, 1971), (Turchin, 1980). This was first implemented in programs by N. Jones and his group (Jones *et al.*, 1985, 1989), see also Romanenko (1988). An implementation of an automatic autoprojector (i.e. self-applicable partial evaluator for a higher order language (a subset of Scheme) is described in (Bondorf, 1990). These transformations involve metasystem transitions. R. Glück (1991) provided examples of multiple metasystem transitions in partial evaluation. For a bibliography on partial evaluation up to 1988 see Sestoft and Sondergaard (1988). Metasystem transition in programming is also referred to as *metaprogramming* (Bundy and Welham, 1981; Bowen and Kowalski, 1983; Takeuchi and Furukawa, 1986; Safra and Shapiro, 1986).

Driving and looping back in supercompilation are close relatives of Burstall-Darlington unfolding and folding, respectively (Burstall and Darlington, 1977), which have been taken up by many researchers (Partsch and Steinbrueggen, 1983). The use of generalization in supercompilation is similar to that in Boyer and Moor (1975).

Several techniques similar to supercompilation appeared lately. 'Online' partial evaluation (Glück, 1991; Weise *et al.*, 1991), as distinct from the 'offline' method,

which includes a preliminary binding-time analysis, is, essentially a variety of supercompilation.

There are many parallels between the program transformation work using functional languages, like we do, and the similar work based on Prolog. In particular, the transformational method in Proietti and Pettorossi (1990, 1991) is an exact analogue of supercompilation carried over from the functional language to Prolog.

2 Refal programs and graphs

The language we use is the same as used in the supercompiler: Refal. A systematic definition of Refal, Refal graphs, and the basic operations on Refal graphs can be found in Turchin (1980, 1986). In the present paper the exposition is based on a few examples. We give a brief definition of Refal programs and graphs (in the volume sufficient for this paper), followed by two examples of supercompilation. The reader is referred to Turchin (1986) for more rigorous and detailed definitions. For more information about Refal, and, in particular, *metacoding* see Turchin (1989).

Consider the following simple program in Refal:

```
* Program 1 (iterative)
<F x> = <Fb [], <Fa [],x>>

<Fa y,'a'x> = <Fa y'b',x>
<Fa y,[]> = y

<Fb y,'b'x> = <Fb y'c',x>
<Fb y,[]> = y
```

Refal is a strict functional language with very simple semantics. A function definition is a list of equations called *sentences*, where in the left side there are function calls for various cases of arguments represented by patterns, and in the right side there are replacements for these cases expressed in terms of the variables which appear in the pattern. Lines starting with an asterisk are comments.

Refal functions are defined on the domain of all *object expressions*, which are composed from symbols and parentheses, with parentheses properly paired according to the usual rules. Symbols are characters, numbers, and symbolic names. Characters and sequences of characters are enclosed in quotes when it is necessary to distinguish them from symbolic names, numbers, and special signs, e.g. `'a'`, `'a+bc'`, `'a'+'bc'`; the last two strings of characters are the same. Symbolic names start with a capital letter, e.g. `F`, `Pred3`, which distinguishes them from variables, which are represented by symbolic names starting with a lower case letter, e.g. `x`, `y3`. Where it is necessary for readability, the metasymbol `[]` is used for the *empty expression*, a string of length zero. Otherwise it is represented by just nothing; thus `()` and `([])` represent the same expression.

An expression which may include variables is referred to as a *pattern*. Thus `(x)` is a pattern expression standing for any expression enclosed in parentheses; `'a'tail` is any expression starting with the character `'a'`.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

Function calls in Refal are delimited by *activation brackets* < and >; <F x,y> is a function F of x and y.

Pattern matching and substitution are the only operations in Refal. We denote the matching of an object expression E to a pattern P as $E : P$. Its result is either a failure or the assignment of some values to the variables in the pattern P. For instance, the matching 'dogs':'d'x's' ends successfully with 'og' as the value of x.

The semantics of Refal is defined by the *Refal machine* which executes Refal programs. The Refal machine works by steps, each step being an application of one sentence, namely the first applicable one (in the order they are listed). A sentence is applicable when its left side can be matched to the argument of the function call. The sentence is used by substituting the right side for the function call as matched to the left side.

In the program above, function Fa goes from left to right through its argument, which is assumed to consist of letters 'a' only, and replaces every 'a' by 'b'. The result is accumulated in the variable y, initially empty. Function Fb works in a similar fashion, transforming a b-string into a c-string.

While a Refal program is convenient to write and read, a better – for automatic program transformation – representation of an algorithm is a *Refal graph*, which is nothing but a graph of states and transitions of a Refal machine with a certain program in its memory and a certain initial *configuration* to be evaluated. A configuration is a general Refal expression which represents the current state of computation. It may include free variables, thus standing for a set of exact states of the machine. The nodes of a Refal graph are configurations. Its edges are operations on configuration variables which result in the jumps from one state to another. Refal graphs can be seen as flow-charts for Refal programs. In fact, it is the language of Refal graphs that is our working language. Refal programs are used only as initial definitions, because they are more convenient for the user.

To show Refal graphs, we convert the definition of the function Fa from Program 1 above into the graph form:

```
[1] = <Fa x,y>

[1] x→'a'; y'b'←y [1]
[+] x→[]; y←out
```

Here the first line is not a part of the graph proper; it establishes the connection between graph terms (numbered configurations) and program terms. When a configuration notation is followed by equality sign, it is its definition; otherwise it is its development, i.e. the graph defining its evaluation.

In our graph, x→'a'x is a *contraction* for x. We use two kinds of operations in Refal graphs, which are special cases of the general operation of matching: *contractions* and *assignments*. A contraction is the matching $v : P$, where v is a variable. It is denoted as $v \rightarrow P$. An assignment is a matching where the pattern is a single variable v; it is denoted as $E \leftarrow v$. In both cases the values of the variables on the left side of the operation are assumed to be known, so that the left side can be reduced to an object expression (i.e. without variables) before the

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

operation. The values of the variables in the right side are defined in the execution of the operation. Contractions combine conditions with computation of values. The contraction $v \rightarrow P$ checks that the value of v can be matched to P, and if so, assigns values to the variables in P. Assignments are unconditional. Both operations can be seen as substitutions for v, which is indicated by the direction of arrows. Variable v belongs to the adjacent configuration: the start of the transition in the case of contraction, and the end of it in the case of assignment.

The contraction x→'a'x includes x on both sides. This means that x is redefined in the operation. If its initial value was 'aaa', it becomes 'aa'. The sign [+] stands for the beginning of another edge from the configuration above (branching). The special variable out stands for the final result of computation.

Now you can read the graph as follows: if x starts with 'a' (x→'a') then add 'b' to y (y'b'←y) and call [1] recursively. If x is empty, the final value of y is assigned to out.

Program 1 does its job by operations on variables in the same recurrent configuration (tail recursion, while loop). Compare it with the following program which displays recursion proper:

```
* Program 2 (recursive)
<F x>      = <Fb <Fa x>>

<Fa 'a'x> = 'b'<Fa x>
<Fa []>    = []

<Fb 'b'x> = 'c'<Fb x>
<Fb []>    = [];
```

With both programs, we would like to do optimization by merging two loops into one. With Program 2 this goal is easily achieved by the supercompiler described in Turchin (1986). We shall give a brief summary of this procedure, referring the reader to that paper for details.

The supercompiler starts with a given initial configuration of the Refal machine, which in our case is a call of function F with an arbitrary argument:

```
[1] = <F x>
```

Now the supercompiler develops the configuration [1], i.e. uses the Refal machine on it in accordance with Program 2. The first step does not depend on the argument x and results in the configuration:

```
[2] = <Fb <Fa x>>
```

The graph at this stage is simply

```
[1] [2]
```

There is an implicit edge here from [1] to [2] which carries no operations: an unconditional transition from one configuration to another.

Even though the Refal machine is defined in the applicative way (the inside-out order of evaluation), and the final result of supercompilation is executed, again,

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

as an applicative program, the supercompiler uses lazy, i.e. outside-in, evaluation, unless it leads to a repeated computation of the same expression. In the cofiguration [2], the supercompiler tries to make a step in the evaluation of Fb, but immediately finds this impossible and switches to the inner function call of Fa.

A step in this call requires the *driving* of the free variable x through the definition of Fa. A branching is created: if x starts with 'a', the first sentence will be used; if it is empty, the second sentence. The result of the step is the graph:

```
[1] [2] x→'a'x [3]
    [+] x→[]; []←out

[3] = <Fb 'b'<Fa x>>
```

The next step of the Refal machine is the computation of the function Fb in [3]. The result is:

```
'c'<Fb <Fa x>>
```

It is a composite expression including a passive part 'c' and an active part, the composition of two function calls identical to [2]. Thus, the configuration [2] reappears, resulting in a recursive loop. The second branch, x→[], produces []. The result is the following program:

```
* Improved Program 2
<F 'a'x> = 'c'<F x>
<F []>   = []
```

The original program required two passes through the argument-strings; the transformed program does the job faster, in one pass. Supercompilation of Program 2 resulted in optimization. But when we try to achieve an analogous transformation by supercompiling the iterative Program 1, we fail.

Indeed, let us try using the same technique as with Program 2. In one unconditional step the starting configuration becomes:

```
[2] = <Fb [],<Fa [],x>>
```

The supercompiler tries to evaluate Fb, but finds it impossible before Fa is evaluated, at least partially. So, one step is made in the evaluation of Fa.The first branch, with the contraction x→'a'x, leads to the configuration:

```
[3] = <Fb [],<Fa 'a',x>>
```

In this configuration, Fb cannot be evaluated, again. Comparing [3] with [2], the supercompiler finds it necessary to generalize [] and 'a' as a new variable y, otherwise the process of driving would be infinite. The generalized configuration becomes:

```
[2g] = <Fb [], <Fa y,x>>
```

Then the supercompiler reduces [2] to [2g] by the assignment []←y:

```
[1] [2] []←y [2g]
```

and develops the state [2g], which loops on itself, so the supercompiler has no

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

choice other than to declare it *basic*, i.e. a necessary part of the final program. Going on along the branch where x→[], it makes another generalization and loops back to the second basic configuration:

```
[3] = <Fb y1,x1>
```

The resulting graph (after merging [1] and [2] and renaming the states) is:

```
* The graph for Program 1
[1] []←y [2] x→'a'x; y'b'←y [2]
          [+] x→[]; y←x1; []←y1 [3]
[3] x1→'b'x1; y1'c'←y1 [3]
[+] x1→[]; y1←out
```

which represents exactly the same algorithm as the original program.

Thus supercompilation fails to make optimization in this case. However, as we shall see in Section 7, with a metasystem transition to computation histories, the same algorithm of supercompilation performs automatically the desired optimization, merging two iterative loops into one.

3 Walk grammars

We introduced a Refal graph as a flow-chart that describes one step of the Refal machine. Now we generalize it to describe *any number* of steps. Then a graph can be seen as a collection of computation histories of various lengths. Let us take the definition of function Fa from Program 1.

```
<Fa y,'a'x> = <Fa y'b', x>
<Fa y, []>  = y
```

The graph for it is:

```
[1] w1 [1]
[+] w2
```

where the transitions (elementary walks) are:

```
w1 =  x→'a'x; y'b'←y
w2 =  x→[]; y←out
```

There are two computation histories of length 1 (in terms of steps): w_1 [1] and w_2. The first is unfinished, the second finished. The unfinished computation can be continued by any of the two elementary walks starting at [1]; we have $w_1 w_1$ [1] and $w_1 w_2$. This can be repeated infinitely, generating all possible finished walks in the original graph. They are of the form $w_1^n w_2$, with all natural numbers n.

By a *class* we shall mean a set expressible as a Refal pattern. For each finite walk w, finished or unfinished, there is a class of input data such that every member of this class, when used as the input data, results in a computation history which is exactly w. We call these classes *input neighborhoods*. Since all elements of a neighborhood have the same computation history, they are, indeed, close to each other in an

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

important way. To take an example, the walk w_1w_1 has the input neighborhood `'aa'x` for the argument x, and y for y (the argument y is arbitrary; the choice of a walk does not depend on it). We also can speak of output neighborhoods, the one in this example being `<Fa y'bb',x>`.

Walks can be simplified by equivalence transformations. The basic formal rules of operating with walks can be found in Turchin (1986). They include *composition* rules, *commutation* rules, and the rule of *clashing*. Here we shall do transformations as we need them, semiformally, in the hope that the validity of the rules is obvious. Thus in the example above we can do the transformation:

$$\begin{aligned} w_1w_1 &= \texttt{x→'a'x; y'b'←y; x→'a'x; y'b'←y [1]} \\ &= \texttt{x→'a'x; x→'a'x; y'b'←y; y'b'←y [1]} \\ &= \texttt{x→'aa'x; y'bb'←y [1]} \end{aligned}$$

We have used a commutation rule in the first step of transformation, and a composition rule in the second. The resulting walk has the form of a Refal graph which in one step does the job that the original program did in two steps. The input neighborhood is as stated above, and so is the output neighborhood after we substitute the assignment for y into `[1]`.

In order to try a given walk with a given input, we simply add the assignments for the input variables at the beginning of the walk, e.g.

```
'aaa'←x; []←y; x→'aa'x; y'bb'←y [1]
```

After the obvious commutation we have a pair of operations:

```
'aaa'←x; x→'aa'x
```

We refer to it as a *clash* of an assignment and a contraction for the same variable. The resolution of a clash is in matching of the assigned value to the pattern in the contraction:

```
'aaa':'aa'x = 'a'←x
```

This clash has been successfully resolved, with `'aa'` as the new value of x. After combining assignments the walk is reduced to:

```
'a'←x; 'bb'←y [1]
```

Suppose we try the same walk, but with `'a'` as the initial value for x. Then we have the clash:

```
'a':'aa'x
```

which has no resolution. With this initial data this walk is unfeasible. It is easy to see that for the finished walk $w_1^3w_2$ there is exactly one value for x with which the walk is feasible, namely `'aaa'`. With all other data, as well as with all other walks for this data, the walk will be unfeasible. This is how the selection of a needed walk takes place.

Unfeasible walks may appear not only because of input data, but also from within the algorithm itself. Take this walk in the graph for Program 1:

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

```
[]←y; x→'a'x; y'b'←y; x→[]; y←x1; x1→[]; y1←out
```

After combining `y'b'←y` and `y←x1`, we have an assignment for `x1` which clashes with the contraction for it:

```
y'b'←x1; x1→[]  =  y'b':[]
```

and is unresolvable. Only those finished walks are feasible where the number of loops in `Fa` is the same as in `Fb`.

We shall be mostly interested in finished walks, so when saying 'walk' we shall mean a finished walk.

The set of all syntactically possible (i.e. without regard for feasibility) walks in a graph can be defined by a context-free grammar where active configurations of the Refal machine become nonterminals, while sequences of contractions and assignments make up terminal symbols. Indeed, each node represents one configuration, the number of different configurations is given in advance and thus is finite, and each node representing the same configuration is the root of the same subgraph. Transitions starting from a configuration $[i]$ become grammar rules for the nonterminal S_i. In particular, the grammar for the graph of `Fa`, if we use the above notation for transitions, is:

$S_1 \Rightarrow w_1 S_1$
$S_1 \Rightarrow w_2$

The grammar for the whole Program 1 (see its graph in Section 2) is:

$S_1 \Rightarrow$ `[]←y;` S_2
$S_2 \Rightarrow$ `x→'a'x; y'b'←y;` S_2
$S_2 \Rightarrow$ `x→[]; y←x1; []←y1;` S_3
$S_3 \Rightarrow$ `x1→'b'x1; y1'c'←y1;` S_3
$S_3 \Rightarrow$ `x1→[]; y1←out`

Subgraphs in walks will be enclosed in square brackets. Thus the grammar for the improved Program 2 is:

$S_1 \Rightarrow S_2$
$S_2 \Rightarrow$ `x→'a'x; [`S_2`]; 'c'out←out`
$S_2 \Rightarrow$ `x→[]; []←out`

The grammar for `Fa` is regular. Therefore, the set of walks it represents can be also rendered by a regular expression. It is $w_1^* w_2$. The grammar for the whole Program 1 is also regular; the grammar for Program 2 is not.

In the rest of this paper we confine ourselves to Refal programs for which the corresponding walk grammar is regular. All sentences in such a program must have a right side which either is completely passive (no function calls, a terminal branch) or is a call of one function, without nested calls. We refer to the subset of Refal which observes these restrictions as *flat* Refal. This subset is algorithmically universal, which follows from the fact that the Universal Turing Machine can be defined in it. Thus every Refal program can be reduced to a program in flat Refal, and our method of program transformation, where complete sets of finished walks are represented by regular expressions, has a full generality.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

4 Metavariables

Variables in Refal expressions are *free*. <F x> stands for a call of F with an unspecified argument x. Variables in Refal graphs are also free, i.e. free to take any values. We shall refer to free variables as variables of *level* 0. In metasystem transitions, variables of other levels emerge. The program Int, which interprets or transforms function F must deal with the graph of F, in particular with its variables and function calls, as with certain fixed objects, pieces of data. At the same time we may wish, when calling Int, to leave some parts of the argument unspecified, i.e. represented by free variables. We must distinguish these free variables from the variables in the graph. We shall say that a variable is at the *reference level*, if it is free. A function call is at the reference level, if it will be evaluated when submitted to the Refal machine. The syntax of variables and function calls, as it is defined in Refal, is the syntax of the reference level.

There are two ways of maintaining the separation between different metasystem levels: we can fix either the reference level, or the starting level.

With the fixed reference level method, which is currently used in the Refal system, the reference level is assigned index 0 and is expressed in terms of the regular Refal syntax. Thus such variables as x are always free by definition. When a metasystem transition takes place, the general expressions which are to become objects of work for a higher level function are *metacoded*, i.e. converted into object expressions using a certain standard procedure. This procedure (metacode) may be different. In the present implementation of Refal, for example, the e-variable x becomes an object expression '*EX'. The variables converted in this way are said to be at the level -1. After one more metasystem transition, this variable, which is now at the level -2, becomes represented by '*VEX'.

With the fixed starting level, metasystem transitions are performed differently. We do not metacode the existing variables, but simply raise the reference level by one, so that it again becomes the top level of the metasystem staircase; if the index of the top level was r it becomes $r + 1$. The variables of level r, which were free before the metasystem transition, are not free any more, but are treated as data. The variables that should be free on the new level $r + 1$ will be referred to as *metavariables*. They must be syntactically distinguishable from regular Refal variables and metavariables of other levels.

While metacodes of variables have negative level indexes, metavariables have positive indexes, but the index of the machine in control (i.e. function being evaluated) is always greater by one than the level of the machine controlled (i.e. function being transformed). The fixed reference level system does not require any extensions of the language Refal as described above, because the metacoded variables can be represented by Refal expressions composed of characters and symbolic names. In contrast, the fixed starting level system requires extension of the basic elements of Refal by a potentially infinite set of special objects: metavariables. The rules of the game must also be updated because the reference level is not fixed any more; it becomes part of the semantics of Refal expressions. Depending on the reference level, different metavariables will be treated as free.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

Both systems of the nomenclature in metasystem transitions can be implemented in the computer and have their advantages and drawbacks. The fixed reference level system requires a metacode transformation of the whole Refal expression in metasystem transitions. With the fixed starting level system, the interpretation of Refal objects depends on the current index of the reference level. In particular, the program we obtain in the end will be expressed in terms of some reference level r, and to convert it into a normal Refal program it will be necessary to perform r metacode transformations (this can be done in one pass, of course). The language Refal can me extended so as to include variables of different levels.

In the present paper we use the fixed starting level system, as it makes it easier for the reader to follow. We shall need only one level of metavariables, and we distinguish them by putting # in front of the variable name, e.g. #x.

5 Graph interpreter

A Refal graph is a product of a Refal program and an initial configuration. It defines a function of the variables which enter the initial configuration: *input* variables. The interpreter `Int` is a function which uses the graph to actually compute the value of the function when the values of the input variables are given. But we define `Int` so that its argument is not literally the recursive Refal graph itself, but a condensed representation of the set (usually infinite) of all *finished walks* in the graph in the form of a regular expression.

As an example, let us take the function `<Fa y,x>` as defined in Program 1:

```
<Fa y,'a'x> = <Fa y'b',x>
<Fa y, []>  = y
```

and the most general call `<Fa y,x>` as the initial configuration. Then the total set of walks is represented by the regular expression L^*w, where

$L =$ `x→'a'x; y'b'←y`
$w =$ `x→[]; y←out`

(From now on, to make the notation easier to grasp, we use L for walks appearing in loops, and w for the others, with subscripts when necessary).

We shall use the sign $+$ to separate alternatives. Our regular expression can be thought of as the infinite sum:

$$w + Lw + LLw + LLLw + \cdots$$

The function `Int` makes an equivalence transformation of a sum of walks. As long as the input variables are *object* expressions (not including variables), the result of this transformation of each walk is either failure, when this walk could not be taken with the given inputs, or the walk of the form E `←out`, where E is the desired value of the call. Function `Int`, as an interpreter, at this moment outputs E as desired result. In the final Refal graph for `<Int ...>` this result must be assigned to the output variable `out`. However, the reference level of `Int` is 1, not 0. The output

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

variable at this level must be written as #out. Thus we have the rule:

$$\texttt{<Int } E\texttt{←out>} = E\texttt{←\#out}$$

The empty result [] represents the absense of feasible walks.

The following properties of additivity and distributivity of the function Int are certain:

$$\texttt{<Int } G_1 + G_2\texttt{>} = \texttt{<Int } G_1\texttt{>} + \texttt{<Int } G_2\texttt{>}$$

$$\texttt{<Int } G(G_1 + G_2)\texttt{>} = \texttt{<Int } GG_1 + GG_2\texttt{>}$$

$$\texttt{<Int } (G_1 + G_2)G\texttt{>} = \texttt{<Int } G_1G + G_2G\texttt{>}$$

where G, G_1, G_2 are any sums of walks (graphs).
Also:

$$\texttt{<Int } G \texttt{ + []>} = \texttt{<Int [] + } G\texttt{>} = \texttt{<Int } G\texttt{>}$$

To compute F for a given set of input values, say 'aa' for x and [] for y, we make the corresponding assignments to the input variables and put it in front of the graph, or rather the regular expression, and then call Int:

$$\texttt{<Int 'aa'←x; []←y; } L^*w\texttt{>}$$

Now all walks start with these assignments (distributivity). Seeing the regular expression as an infinite sum, we can compute Int for each walk:

<Int 'aa'←x; []←y; w> = []
<Int 'aa'←x; []←y; Lw> = []
<Int 'aa'←x; []←y; LLw> = 'bb'←#out
<Int 'aa'←x; []←y; $LLLw$> = []
...

All the remaining walks produce [], so the final outcome is 'bb'. If the Refal program were non-deterministic, the result of Int could be a sum of final assignments, possibly infinite.

With inputs given without indeterminacy, Int can be defined as a straightforward procedure executing the operations on walks from left to right. This way, however, will not necessarily be the most efficient. For example, if there is a definitely impossible clash (like 'a':'b') inside a walk, there is no sense to execute the preceding operations; the result will be [] anyway. What is more important, such a walk may include undefined elements, and the result still may be determined. Sometimes it may be possible to get a speedy answer by examining the walk from the end, or transposing and combining various operations in it.

Thus, function Int is, essentially, a set of equivalence transformation rules; the order in which they are used may vary according to the strategy chosen, which may be further adjusted to current needs by the supervising program of the supercompiler. In the present paper we do not formally define our strategy of transformation; we shall only formulate two general guiding principles on which it is based. The author is confident that using these principles it would not be difficult to write a program of automatic transformation sufficient for all the examples presented here, and more.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

But of course, this statement will be proven only when such a program is actually written.

- The first principle is the usual method of recursion: pick up pieces of the arguments (i.e. walk operations), combine them using equivalence relations, and try to reduce what remains to the same form as was there before the separation of the pieces.
- The second principle is to pick up pieces in such a manner that *clashes*, appear between them. The reason for this rule is that in transformations of walks it is clashes that represent steps of the Refal machine. The general structure of an elementary walk, i.e. the one desribing one step of the Refal machine, is this: a series of contractions for the variables of the old configuration, followed by a series of assignments for the variables of the new configuration. When one walk is followed by another, the assignments of the first walk clash with the contractions of the second. When we chain walks according to the grammar of a Refal graph, we only schedule computing operations, without actually performing them. We perform computation when we resolve clashes and make substitutions. The more clashes we have resolved, the greater is the part of computation that we have performed during program transformation.

When we use regular expressions to represent sets of walks, there are two ways to chip off a piece: from the beginning and from the end. Accordingly, we introduce two rules of handling regular expressions to be used by the interpreter `Int`. We call them *unwinding* rules, and they can be applied independently to any loop (closure, Kleene star) in the regular expression.

(a) Left unwinding: $G_1L^*G_2 = G_1LL^*G_2 + G_1G_2$
(b) Right unwinding: $G_1L^*G_2 = G_1L^*LG_2 + G_1G_2$

Here G_1 and G_2 are arbitrary regular expressions. The choice of one of the rules is defined by the second general principle mentioned above. If G_1L contains a clash (which will be the case if G_1 includes an assignment for some variable, while the loop L includes a contraction for the same variable), then left unwinding should be tried. If LG_2 has a clash, this is a reason to use right unwinding. It is possible, of course, that both operations result in a clash.

The simplest interpretation strategy is a direct interpretation in applicative order. It works as follows. We start from the input assignments at the beginning of the graph, scan the walks and use the commutation and composition rules to produce clashes, which are then resolved. As we meet a loop, we unwind it from the left, and again resolve the clashes. Then we try to reduce the resulting regular expression to the one we started with (recursion).

As we mentioned before, this process would be infinite, if not for the supercompiler on the next metasystem level, which oversees and controls the work of the interpreter. The supercompiler should notice the recursion and build up a finite recursive program corresponding to the walk set in the argument of `Int`.

We discussed the work of `Int` when a specific input is given. But we are really interested, of course, in the case where the inputs are not specified, or specified

partially. These gaps in the input specification will be represented by metavariables. For example, if we want to leave both arguments of Fa unspecified, i.e. to transform the function Fa as such, we must supercompile:

```
<Int #x←x; #y←y; G>
```

where G is the regular expression for <Fa x,y>.

6 The method

Now we summarize our method of program transformation and apply it, for illustration, to the identical transformation of the function <Fa x,y>. The purpose is to show how the technique works in a very simple case, and check that the identical transformation is among those the technique allows. This is not quite trivial, because identical transformation with our method is not just rewriting the definition. The program must reproduce itself after a double metasystem transition.

Step 1. Given a Refal program and an initial configuration, construct a Refal graph for it. In our example, <Fa x,y>, it is:

```
[1] x→'a'x; y'b'←y [1]
[+] x→[]; y←out
```

Step 2. Convert the graph into a context-free grammar:

$S_1 \Rightarrow$ `x→'a'x; y'b'←y;` S_1
$S_2 \Rightarrow$ `x→[]; y←out`

Step 3. Assuming that the grammar is regular, make it into a regular expression:

```
(x→'a'x; y'b'←y)* x→[]; y←out
```

Step 4. For each input variable, add the assignment of a metavariable to it at the beginning of the regular expression, and form a call of the interpreter Int with the resulting expression as the argument:

<Int $w_1 L^* w_2$>

where

$w_1 =$ `#x←x; #y←y`
$L =$ `x→'a'x; y'b'←y`
$w_2 =$ `x→[]; y←out`

Step 5. Supercompile the call of Int, using the properties of Int and the two basic principles (recursion and clashes) formulated in Section 5.

Let us see the process of supercompilation in our example. The initial configuration is:

[1] = <Int $w_1 L^* w_2$>

How should we proceed? How to use unwinding rules? In search of clashes, put the

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

finger at the first assignment, which is #x←x in w_1. Move to the right looking for a contraction for x. We find it in the loop walk L. Therefore, to produce a clash, we must unwind L *on the left*. After doing so we have:

<Int $w_1LL^*w_2 + w_1w_2$>

By the additivity of Int, mimicked by the additivity of Scp (because both functions are graph transformers), the graph under supercompilation becomes:

[1] [2]
[+] [3]

[2] = <Int $w_1LL^*w_2$>
[3] = <Int w_1w_2>

Consider the configuration [2]. The loop breaks the original regular expression into two segments. Because of the left unwinding, the first segment has become:

$$w_1L = \#x{\leftarrow}x;\ \#y{\leftarrow}y;\ x{\rightarrow}'a'x;\ y'b'{\leftarrow}y$$

Now it will be transformed by the interpreter. Using the commutation and composition rules yields:

$$w_1L = \#x{\leftarrow}x;\ x{\rightarrow}'a'x;\ \#y'b'{\leftarrow}y\#x{:}'a'x;\ \#y'b'{\leftarrow}y$$

Then the clash #x:'a'x for x must be resolved. We must remember that the metavariable #x is not a graph variable, but a free variable of the interpreter, which will be replaced by a specific object expression (constant) whenever the interpreter is running. For example, if #x is 'aaa' then the walk starts with 'aaa'←x. After the resolution of the clash it becomes 'aa'←x. The clash is resolved successfully only if the value of #x starts with 'a'. The supercompiler, which drives the call of Int, will create a walk with the contraction #x→'a'#x. The resolution of the clash, namely:

$$\#x{:}'a'x = \#x{\rightarrow}'a'\#x;\ \#x{\leftarrow}x$$

looks as if #x and x were of the same metasystem level. The difference is only in where the contraction belongs: it belongs to the graph being under construction, not to the argument of Int. The graph under supercompilation becomes:

[1] [2] #x→'a'#x [4]
[+] [3]

[4] = <Int #x←x; #y'b'←y; L^*w_2>

Configurations such as [2], which replace the preceding configuration unconditionally and without any operations, are referred to as *transient*. There is no need to keep transient configurations in the graph. Below we shall omit them without a notice.

Remembering what w_1 stands for, we can rewrite [1] as:

[1] = <Int #x←x; #y←y; L^*w_2>

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

We see then that [4] is a special case of [1]. The supercompiler notices this too and reduces it to [1] by the assignment #y'b'←#y:

```
[1] #x→'a'#x; #y'b'←#y [1]
[+] [3]
```

Now we transform the loop exit [3]:

$$\begin{aligned} \texttt{<Int } w_1 w_2\texttt{>} &= \texttt{<Int \#x←x; \#y←y; x→[]; y←out>} \\ &= \texttt{<Int \#x:[]; \#y←out>} \\ &= \texttt{\#x→[] <Int \#y←out>} \\ &= \texttt{\#x→[]; \#y←\#out} \end{aligned}$$

This is the end of supercompilation. The result is:

```
[1] #x→'a'#x; #y'b'←#y [1]
[+] #x→[]; #y←#out
```

which is the same as the original graph for Fa, except that it is expressed, as we anticipated, in terms of metavariables. When Scp is written as a Refal program, the graphs and regular expressions with which it works will be all downgraded in the metacode, and the output program will be in terms of normal variables.

7 Merging iterative loops

Now we want to improve Program 1 for <F x>. As we saw above, direct supercompilation leaves it unchanged. We will show that with the use of metasystem transitions the transformation becomes possible.

7.1 Stage 1: simultaneous unwinding

The graph for Program 1 is represented as the following regular expression:

$$G = w_1 L_a^* w_2 L_b^* w_3$$

where

$$\begin{aligned} w_1 &= \texttt{\#x←x; []←y} \\ L_a &= \texttt{x→'a'x; y'b'←y} \\ w_2 &= \texttt{x→[]; y←x1; []←y1} \\ L_b &= \texttt{x1→'b'x1; y1'c'←y1} \\ w_3 &= \texttt{x1→[]; y1←out} \end{aligned}$$

Note that we have already included the initial assignment in w_1.

Now let us see the process of supercompilation of the interpreter. The initial configuration is:

$$\texttt{[1] = <Int } w_1 L_a^* w_2 L_b^* w_3 \texttt{ >}$$

If we apply the same strategy as in Section 6, i.e. unwind on the left one loop at a

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

time, then our method will return the original program. In order to merge the two loops, we shall unwind them *simultaneously*. We must unwind both loops on the left, because in this way we produce clashes, as required by our second principle of walk transformation.

Simultaneous unwinding breaks the set of walks in four parts:

$$\begin{aligned}\texttt{[1]} &= \texttt{<Int}\ G_1 + G_2 + G_3 + G_4\texttt{>} \\ &= \texttt{<Int}\ G_1\texttt{>} + \texttt{<Int}\ G_2 + G_3 + G_4\texttt{>}\end{aligned}$$

where

$$\begin{aligned}G_1 &= w_1 L_a L_a^* w_2 L_b L_b^* w_3 \\ G_2 &= w_1 L_a L_a^* w_2 w_3 \\ G_3 &= w_1 w_2 L_b L_b^* w_3 \\ G_4 &= w_1 w_2 w_3\end{aligned}$$

Two loops break G_1 into three segments. The first and the second are transformed as follows:

$$\begin{aligned}w_1 L_a &= \texttt{\#x←x; []←y; x→'a'x; y'b'←y} \\ &= \texttt{\#x→'a'\#x; \#x←x; 'b'←y} \\ w_2 L_b &= \texttt{x→[]; y←x1; []←y1; x1→'b'x1; y1'c'←y1} \\ &= \texttt{x→[]; y→'b'y; y←x1; 'c'←y1}\end{aligned}$$

The third segment, $L_b^* w_3$, is unchanged. The supercompiler compares the transformed G_1 with the initial state `[1]`. As in Section 6, `#x→'a'#x` is left in front of the call of `Int`. The supercompiler compares `<Int` G_1`>` with `<Int` G`>`. It compares $w_1 L_a$ with w_1:

```
#x←x;   []←y
#x←x;  'b'←y
```

and sees that the new configuration – in that part – is not a subset of the old one. It constructs the generalized segment

$w_1{}^g$ `= #x←x; #y←y`

by generalizing `[]` and `'b'` as the metavariable `#y`.

Now $w_2 L_b$ is compared with w_2:

```
x→[];            y←x1;   []←y1
x→[]; y→'b'y;  y←x1;  'c'←y1
```

The expressions `[]` and `'c'` are generalized as a metavariable `#y1`. But there is one more operation in $w_2 L_b$ as compared with w_2: the contraction `y→'b'y`. The absence of this contraction is interpreted as the identical contraction `y→y`. Generalizing it with `y→'b'y`, we come to `y→#y2 y`, where `#y2` is one more metavariable. The generalized second segment is:

$w_2{}^g$ `= x→[]; y→#y2 y; y←x1; #y1←y1`

Let us sum up what happened up to this moment. We were driving `<Int` G`>`. After

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

unwinding, <Int G> became a sum of four configurations. Then G_1 was driven. It was transformed into something close to G, which required a generalization. Now <Int G> must be reduced to the generalization, which is:

<Int G^g> = <Int $w_1{}^g L_a^* w_2{}^g L_b^* w_3$>

The reduction is done by assigning [] to the three metavariables which appeared in the generalization:

```
[1] []←#y; []←#y2; []←#y1 [2]
[2] = <Int G^g>
```

The former development (the result of driving) of [1], including the breakdown into four components, is discarded, and the driving of [2] starts.

The next stage of supercompilation is similar to the beginning, but $w_1{}^g$ and $w_2{}^g$ replace w_1 and w_2. Two loops are unwound on the left and the component $\dot{G}_1{}^g$ is transformed. Unlike the beginning stage, however, it reduces to G^g. The graph becomes:

```
[1] []←#y; []←#y2; []←#y1 [2]
  [2] #x→'a'#x; #y'b'←#y; #y2'b'←#y2; #y1'c'←#y1 [2]
  [+] <Int G2^g + G3^g + G4^g>
```

Now we drive <Int $G_2{}^g$>, where

$$G_2{}^g = w_1{}^g L^a L_a^* w_2{}^g w_3$$

```
    = #x←x;#y←y; x→'a'x; y'b'←y L*a
        x→[]; y→#y2 y; y←x1; #y1←y1; x1→[]; y1←out
```

One loop breaks it in two segments. The first segment is transformed in the now familiar way:

```
#x→'a'#x <Int #x←x; #y'b'←y L*a...>
```

In the second segment, the clash for x1 results in y→[], which combines with the contraction for y. The last two assignments also combine. The branch for <Int $G_2{}^g$> becomes:

```
[+] #x→'a'#x [3]

[3] = <Int #x←x; #y'b'←y L*a x→[]; y→#y2; #y1←out>
```

Now [3] is developed by unwinding L_a^* on the left, so as to clash the contractions for x and y in L_a^* with the assignments to them at the beginning of the walk. Denoting the two segments in [3] by w_1 and w_2 again, we have:

$$w_1 L_a L_a^* w_2 + w_1 w_2$$

The first walk after simplification yields:

```
#x→'a'#x; #x←x; #y'bb'←y; L*a w2
```

which loops back to [3].

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

The second walk, $w_1 w_2$, becomes:

```
#x→[]; #y'b':#y2; #y1←out
```

Now the graph for [3] is:

```
[3] #x→'a'#x; #y'b'←#y [3]
[+] #x→[] #y'b':#y2; #y1←out
```

The subgraph G_3^g is treated in a similar way. Finally, $G_4{}^g$ produces one more walk on the Int level:

$$
\begin{aligned}
w_1{}^g w_2{}^g w_3 \;=\;& \texttt{\#x←x; \#y←y; x→[]; y→\#y2 y;} \\
& \texttt{y←x1; \#y1←y1; x1→[]; y1←out} \\
=\;& \texttt{\#x→[]; \#y:\#y2; \#y1←out}
\end{aligned}
$$

Collecting all four branches, we come to the following total graph resulting from supercompilation:

```
[1] []←#y; []←#y2; []←#y1 [2]
   [2] #x→'a'#x; #y'b'←#y; #y2'b'←#y2; #y1'c'←#y1 [2]
   [+] #x→'a'#x [3] #x→'a'#x; #y'b'←#y [3]
                [+] #x→[]; #y'b':#y2; #y1←#out
   [+] #x→[] [4] #y2'b'←#y2; #y1'c'←#y1 [4]
             [+] #y:#y2'b'; #y1'c'←#out
   [+] #x→[]; #y:#y2; #y1←#out
```

This algorithm is very far from the efficient program we wanted to obtain. At first sight, it is much worse than the original algorithm. Looking more carefully, however, we notice that we have already achieved something in the line of our goal. The main loop of the program, from [2] to itself, starts with chipping 'a' from the input #x and adding 'c' to #y1, which ultimately becomes the output. So we *did* merge the loops, after all.

However, in addition to input and output, the program uses two variables: #y and #y2, which serve as synchronous 'counters' in the two loops. They both start from empty and are lengthened by one 'b' in the main loop. In the last, exit, branch their values are compared in the matching #y:#y2, which, of course, must always succeed.

There are two more loops, at [3] and [4], the exits from which include matchings incompatible with the equality of #y and #y2. These loops can be run infinitely without producing feasible finished walks. They describe walks where the number of cycles in one loop is less or greater than in the other, and such walks are unfeasible. If we perform a careful analysis of operations in these loops, we must come to the efficient program. But how to do this?

7.2 Stage 2: eliminating infinite loops

An important feature of our method is that the same transformation technique can be applied repeatedly and each time bring non-trivial results. So, we simply

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

make another metasystem transition. Metavariables in our graph become normal variables; the variable x is assigned a new metavariable #x. Introducing notation for segments, we reduce the graph to this:

```
[1] w1 [2] L1 [2]
       [+] w2 [3] L2 [3]
              [+] w3
       [+] w4 [4] L3 [4]
              [+] w5
       [+] w6
```

where

w_1 = #x←x; []←y; []←y2; []←y1
L_1 = x→'a'x; y'b'←y; y2'b'←y2; y1'c'←y1
w_2 = x→'a'x
L_2 = x→'a'x; y'b'←y
w_3 = x→[]; y'b':y2; y1←out
w_4 = x→[]
L_3 = y2'b'←y2; y1'c'←y1
w_5 = y:y2'b'; y1'c'←out
w_6 = x→[]; y:y2; y1←out

The regular expression corresponding to this graph is:

$$w_1 L_1^* (w_2 L_2^* w_3 + w_4 L_3^* w_5 + w_6)$$

The first two terms of the sum are those to disappear. Let us take the first term:

$$w_1 L_1^* w_2 L_2^* w_3$$

Our goal is to prove that this regular expression is unfeasible. When we only want to choose between the two alternatives, feasible an unfeasible, without building an equivalent graph, we can simplify the original graph by leaving out all the operations which do not take place in clashes. Thus we can cancel assignments at the end of the walk and contractions at its beginning. So, we cancel y1←out in w_3. Moreover, in the loop L_1 we can throw away the contraction for x, because there are no assignments for x before the loop, and the assignment for y1, because there are no contractions for it after the loop. For the same reason the whole walk w_2 disappears, as well as the contraction for x in L_2 and w_3. We now have the regular expression:

$$w_1 L_1^* L_2^* w_3$$

where

w_1 = #x←x; []←y; []←y2; []←y1
L_1 = y'b'←y; y2'b'←y2
L_2 = y'b'←y
w_3 = y'b':y2; y1←out

It must be stressed that all these simplifications are done exclusively for making the

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

volume of expressions smaller, as we compute by hand. It is not necessary to do this in a computer program.

We do right unwinding in both loops (but sequentially), to produce clashes in w_3 with updated variables y and y2:

```
[1] [2]
[+] [3]
```

[2] = <Int $w_1 L_1^* L_2^* L_2 w_3$>
[3] = <Int $w_1 L_1^* w_3$>

Further computation goes on as follows:

$$L_2 w_3 = \texttt{y'b'}\leftarrow\texttt{y;}\ \ \texttt{y'b':y2 = y'bb':y2}$$

Compare and generalize:

$$w_3{}^g = \ \texttt{y \#y'b':y2}$$

Reduce and redefine:

```
[1] []←#y [2] [3]
          [+] [4]
```

[3] = <Int $w_1 L_1^* L_2^* L_2 w_3{}^g$>
[4] = <Int $w_1 L_1^* w_3{}^g$>

Simplifying $L_2 w_3{}^g$ we find that [3] is reduced to [2] with the assignment 'b'#y←#y. Now [4] becomes:

```
[4] [5]
[+] [6]
```

[5] = <Int $w_1 L_1^* L_1 w_3{}^g$>
[6] = <Int $w_1 w_3{}^g$>

Compute [5]:

$$L_1 w_3{}^g = \ \texttt{y'b' \#y'b':y2'b' = y'b' \#y:y2}$$

Compare it with $w_3{}^g$. To determine if the former is a subclass of the latter, the supercompiler Scp will use the generalized matching algorithm (GMA, see (Turchin, 1986)), matching

$$\texttt{(y'b'\#y:y2):(y \#y'b':y2) = y'b'\#y:y \#y'b'}$$

According to GMA, this will cause a branching with two contractions:

```
[5] #y→#y'b' [7]
[+] #y→[] [8]
```

[7] = <Int $w_1 L_1^*$; y'b'#y'b':y2>
[8] = <Int $w_1 L_1^*$; y'b':y2>

Now [7] is reduced:

```
[7] 'b'#y←#y [4]
```

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

Compute [8]:

```
[8]  [9]
[+]  [10]
```

[9] = $w_1 L_1^* L_1$; y'b':y2
[10] = w_1; y'b':y2

Compute [9]:

$$L_1;\ \texttt{y1'b':y2} = \texttt{y'b'}\leftarrow\texttt{y;\ y2'b'}\leftarrow\texttt{y2;\ y1'b':y2}$$
$$= \texttt{y'bb':y2'b'} = \texttt{y'b':y2}$$

We see here a somewhat unusual kind of recursion, when a configuration is reduced to itself without any assignments:

```
[8]  [9]  [8]
[+]  [10]
```

Computing [10] we see that it includes the failing matching: #y:[]; hence this branch disappears. We still have one more terminal branch left behind: [6]. It also disappears. The final result is a graph where there is not a single terminal branch:

```
[1]←#y  [3]'b'#y←#y  [3]
        [+]  [4]  #y→#y'b';  'b'#y←#y  [4]
             [+]  #y→[]  [8]  [8]
```

This graph produces an empty set of walks, hence the original regular expression $w_1 L_1^* w_2 L_2^* w_3$ is proven, as we anticipated, empty. In a similar manner, computation proves that $w_1 L_1^* w_4 L_3^* w_5$ is also empty. We are left with $w_1 L_1^* w_6$, where

w_1 = #x←x; []←y; []←y2; []←y1
L_1 = x→'a'x; y'b'←y; y2'b'←y2; y1'c'←y1
w_6 = x→[]; y:y2; y1←out

The program corresponding to this regular expression still keeps updating the counters y and y2, only to check in the end that they have equal values. We want to get rid of this unnecessary work, so we go on with supercompilation.

For the same reasons as before, we unwind the only loop on the right:

```
[1]  [2]
[+]  [3]
```

[2] = <Int $w_1 L_1^* L_1 w_6$>
[3] = <Int $w_1 w_6$>

$$L_1 w_6 = \texttt{x}\rightarrow\texttt{'a';\ y:y2;\ y1'c'}\leftarrow\texttt{out}$$

Comparing this with w_6, we generalize:

$$w_6{}^g = \texttt{x}\rightarrow\texttt{\#x2;\ y:y2;\ y1\ \#y1}\leftarrow\texttt{out},$$

then make a reduction and again unwind on the right:

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

```
[1] []←#x2; []←#y1 [2] [3]
                   [+] [4]
```

[3] = <Int $w_1 L_1^* L_1 w_6{}^g$>
[4] = <Int $w_1 w_6{}^g$>

Now $L_1 w_6{}^g$ is easily reduced to $w_6{}^g$:

```
[3] 'a'#x2←#x2; 'c'#y1←#y1 [3]
```

Computing [4] we have:

$w_1 w_6{}^g$ = `#x←x; []←y;[]←y2;[]←y1; x→#x2; y:y2; y1#y1←out`
= `#x←x;x→#x2; #y1←out`

Thus we have the graph:

```
[1] []←#x2; []←#y1 [3] 'a'#x2←#x2; 'c'#y1←#y1 [3]
                   [+] #x:#x2; #y1←#out
```

And still this is not the efficient algorithm we want! We have eliminated two counters, y and y2, but only at the price of introducing a new counter x2.

7.3 Stage 3: eliminating non-determinism

Now we shall use the same transformation technique once again. This problem can be seen independently of the loop merging problem; it has its own characteristic feature.

The graph to transform is:

```
[1] []←x2; []←y1 [3] 'a'x2←x2; 'c'y1←y1 [3]
                 [+] x:x2; y1←out
```

Note that this algorithm is *non-deterministic*. The looping branch is unconditional and can be chosen at each step; if we always chose it, the process would be infinite. But we can check the other branch at each step too, and at a certain moment terminate computation with a correct result.

The Refal machine is deterministic, and so are the Refal graphs which are obtained by translating Refal programs. But when we unwind walk loops on the right, we read the walks in the reversed order. Thus a graph obtained in this way will be, generally, non-deterministic. We can use non-deterministic graphs either as an intermediate stage of transformation, or as the desired result, as in the case of function inversion.

Let us transform our algorithm into an equivalent deterministic one. The regular expression corresponding to our graph is $w_1 L^* w_2$, where:

w_1 = `#x←x; []←x2; []←y1`
L = `'a'x2←x2; 'c'y1←y1`
w_2 = `x:x2; y1←out`

Among the rules of operations on regular expressions, there is one which immediately follows from the basic rules: If A is an assignment, and G any regular

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

expression which involves no variables involved in A, then $AG = GA$; if C is a contraction, and G an expression meeting the same requirement, then $GC = CG$. Using this rule, we move `#x←x` from w_1 to w_2 and merge it with the matching, which results in `#x:x2`.

Since by this time the reader should have become familiar with our technique, we shall not write out intermediate graphs, but only highlight operations on regular expressions.

After a right unwinding we have:

$$\begin{aligned} Lw_2 &= \texttt{'a'x2←x2; 'c'y1←y1; \#x:x2; y1←out} \\ &= \texttt{\#x:'a'x2; 'c'y1←out} \\ &= \texttt{\#x→'a'\#x; \#x:x2; 'c'y1←y1} \end{aligned}$$

Generalization results in:

$$w_2{}^g = \texttt{\#x:x2; \#y1 y1←out}$$

and $Lw_2{}^g$ reduces to $w_2{}^g$ by `#y1'c'←#y1`. The termination branch is:

$$\begin{aligned} w_1 w_2{}^g &= \texttt{[]←x2; []←y1; \#x:x2; \#y1 y1←out} \\ &= \texttt{\#x→[]; \#y1←out} \end{aligned}$$

Now we have come to the desired final result:

```
[1] []←#y1 [2] #x→'a'#x; #y1'c'←#y1 [2]
           [+] #x→[]; #y1←out
```

Even though the original algorithm is non-deterministic, it can be executed on a sequential deterministic machine without an exponential explosion. If n is the lengths of the string, the time complexity for this algorithm, because of repeated comparisons `x:x2`, is $\mathcal{O}(n^2)$. But if we have more than one branch in the loop, the deterministic time complexity will be exponential, and our transformation method will still convert it into a linear complexity:

```
[1] []←x; []←y [2] x'a'←x; y'b'←y [2]
               [+] x'b'←x; y'a'←y [2]
               [+] #x:x; y←out
```

This function builds up `x` by adding `'a'` or `'b'` in each step, while `y` is augmented by `'b'` or `'a'`, respectively. If `x` happens to be equal to the input `#x`, then `y` becomes the output.

The regular expression for this problem is

$$w_1(L_a + L_b)^* w_2$$

where

$$\begin{aligned} w_1 &= \texttt{[]←x; []←y} \\ L_a &= \texttt{x'a'←x; y'b'←y} \\ L_b &= \texttt{x'b'←x; y'a'←y} \\ w_2 &= \texttt{\#x:x; y←out} \end{aligned}$$

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

The right unwinding yields:

$$\begin{aligned} w_1(L_a + L_b)^* w_2 &= w_1(L_a + L_b)^*(L_a + L_b)w_2 + w_1 w_2 \\ &= w_1(L_a + L_b)^* L_a w_2 + w_1(L_a + L_b)^* L_b w_2 + w_1 w_2 \end{aligned}$$

It is easy to check that proceeding in the fashion as above we come to the efficient deterministic algorithm:

```
[1] []←y [2] x→x'a'; 'b'y←y [2]
         [+] x→x'b'; 'a'y←y [2]
         [+] x→[]; y←out
```

8 Function inversion

We take a function of one variable:

```
<F x> = <Fa [],x>
```

with Fa defined, again, as in Program 1. We want to invert it. The transformed function should expect a string of letters 'b' as its input and convert every 'b' back into 'a'.

The graph for F is:

```
[1] []←y [2] x→'a'x; y'b'←y [2]
[+] x→[]; y←out
```

To compute the value, or the values, of the inverted function for some input value E, the interpreter must find the walks which end with the assignment of such an expression to out, that after all substitutions it becomes identical to E. This is nothing but the contraction out→E imposed on the output variable out. In particular, when we want a fully inverted function, we must allow for E to be arbitrary, that is to be represented by a metavariable.

Further, the value of the inverse function of a function with one variable x will be the input value of x required in the walks we discovered. The required value for x is the one to which x is contracted, and this value must become the output of the inverse function. Thus we have the rule:

$$\texttt{<Int x→}E\texttt{>} = E \leftarrow \texttt{\#out}$$

If there are more than one qualifying path, there will be more than one assignments to out possible (a non-deterministic program). If the original function is of two variables x and y, then the value of the inverse function will be a pair, and we shall use the rule:

$$\texttt{<Int x→}E_1\texttt{; y→}E_2\texttt{>} = (E_1, E_2) \leftarrow \texttt{\#out}$$

Let the input value for the inverse function be represented by #y. When we add the contraction out→#y to the terminal branch in the graph for F, the resulting clash is resolved as follows:

```
y←out; out→#y = y:#y = y→#y
```

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

Keep in mind that the clash is resolved on the graph level (performed by `Int`), where `y` is a variable and `#y` a constant. If it were on the `Int` level (performed by `Scp`), where `y` is an object expression (the metacode of a variable), and `#y` a variable, the resolution of the clash would be `y←#y`.

All said, the total set of walks is represented by the regular expression $w_1 L^* w_2$, where

$$\begin{aligned} w_1 &= \texttt{[]←y} \\ L &= \texttt{x→'a'x; y'b'←y} \\ w_2 &= \texttt{x→[]; y→\#y} \end{aligned}$$

Naturally, we are going to unwind the loop on the right:

$$w_1 L^* w_2 = w_1 L^* L w_2 + w_1 w_2$$

A formal reason for that, to be used in writing the algorithms, is a consideration of clashes. We want to involve metavariables (data) into clashes. The only metavariable we have is in the contraction for `y` in w_2, while L includes an assignment for `y`. Thus Lw_2 makes a useful clash, while $w_1 L$ achieves nothing.

As in Section 6, we have the following graph under supercompilation:

```
[1] [2]
[+] [3]

[2] = <Int w1L*Lw2>
[3] = <Int w1w2>
```

The transformation follows:

$$\begin{aligned} Lw_2 &= \texttt{x→'a'x; y'b'←y; x→[]; y→\#y} \\ &= \texttt{x→'a'; y'b':\#y} \\ &= \texttt{x→'a'; \#y→\#y'b'; y→\#y} \end{aligned}$$

As in Section 6, the contraction for the metavariable `#y` belongs to the `Scp` level:

```
[1] #y→#y'b' [4]
[+] [3]

[4] = <Int w1L*; x→'a'; y→#y>
```

Unlike Section 6, however, the new configuration `[4]` is not a subclass of `[1]`. The supercompiler compares them:

```
[1] = <Int w1L*; x→[];  y→#y>
[4] = <Int w1L*; x→'a'; y→#y>
```

It generalizes `[]` and `'a'` as a new metavariable `#x`, which results in the generalized configuration, to which `[1]` is reduced by an assignment for `#x`:

```
[1] []←#x [2]

[2] = <Int w1L*; x→#x; y→#y>
```

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

All previous development for [1] is destroyed, and the development process goes on from the (redefined) configuration [2].

Denoting the generalized second segment as

$$w_2{}^g = \texttt{x}\rightarrow\texttt{\#x; y}\rightarrow\texttt{\#y}$$

we have the regular expression $w_1L^*w_2{}^g$ for [2]. We again unwind it on the right and come to the graph:

```
[1] []←#x [2] [3]
          [+] [4]
```

[3] = <Int $w_1L^*Lw_2{}^g$>
[4] = <Int $w_1w_2{}^g$>

The computation of $Lw_2{}^g$ produces, again, a contraction for #y, and:

$$\texttt{x}\rightarrow\texttt{'a'\#x; y}\rightarrow\texttt{\#y}$$

which, this time, reduces to the preceding configuration [2] by 'a'#x←#x:

```
[1] []←#x [2] #y→#y'b';'a'#x←#x [2]
          [+] [4]
```

Finally, we compute [4]

<Int $w_1w_2{}^g$> = <Int []←y; x→#x; y→#y>
#y→[]; <Int x→#x> = #y→[]; #x←out

The inverse function has the graph:

```
[1] []←#x [2] #y→#y'b';'a'#x←#x [2]
          [+] #y→[]; #x←#out
```

If we invert the function Fa of two arguments and agree to represent its output as a pair (x,y) then we come to a correct but, of course, non-deterministic, algorithm:

```
[1] []←#x [2] #y→#y'b';'a'#x←#x [2]
          [+] (#x,#y)←#out
```

Our technique of inversion is close to the interpretive inversion of Refal programs developed by Romanenko (1991). The main difference is that we immediately couple it with the supercompiler.

9 Conclusions

1. We have developed a new method of program transformation which is based on supercompilation, but includes a *metasystem transition*: the supercompiler Scp is applied not to the function F to be transformed, but to an interpreter Int which works with computation histories according to F. Thus Int is a metafunction with regard to F. The examples we presented show the application of this method to various kinds of transformation problems. The method is not targeted at any specific property of the program to transform, and yet produces a variety of optimizations.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

2. A remarkable feature of this method is that it can be applied to the program that has just been transformed and still produce a new, better, result. In Section 7 we transformed the initial program and merged two algorithmic loops into one, but the resulting program was still far from perfect: it was non-deterministic and contained unfeasible branches. After the second transformation these branches disappeared, but the program was still non-deterministic. It is after the third transformation that we obtained the final efficient program. This is in a sharp contrast with the idempotency of most methods of optimization, when applying the same technique more than once gives, essentially, the same result as one transformation. In particular, direct supercompilation is idempotent. We explain this feature of our method by the metasystem transitions which take place between repeated uses of the method. Indeed, when Scp is applied directly to F, the result is still a transformed F. In the subsequent use we apply Scp to the same function. With a metasystem transition, we each time apply Scp to a different function, namely to the metafunction of the preceding function. This is seen clearly if we stick to the fixed starting level method of the representation of metasystem transitions (see Section 4). At the beginning, F is at level 0. When Scp is applied directly, it becomes level 1, and the reference level is 1. The output of Int, which is the new graph for F, is again at level 0. With our new method, Int is at level 1, and Scp is at level 2. Its output is at level 1, not 0; it is expressed in terms of metavariables. If we simply went on repeating the transformation, then the level of the function under transformation would become 2, 3,... etc.: a potentially infinite *metasystem stairway*. However, after each transformation we reduced the level of the outcome, returning the function definition to level 0, this being only the matter of notation. Thus the metasystem stairway, as we used it, was, essentially, reduced to the three levels: 0,1, and 2.

3. There is an intriguing question here: how much can the power of the method be enhanced by consecutive multiple-level metasystem transitions, in particular, the next after the one we used, i.e.:

$$\mathtt{Scp} \succ \mathtt{Int}_2 \succ \mathtt{Int}_1 \succ \mathtt{F} \succ \mathtt{D}?$$

On one hand, when we did this transition for the first time, i.e. from F to $\mathtt{Int}_1$, the power of the method increased greatly. Thus treating $\mathtt{Int}_1$ as F before and adding $\mathtt{Int}_2$ we may hope to obtain even better results. On the other hand, our success has been due to the fact that we used a 'clever' version of Int. Then we could reason that applying the same Int to itself will not really enhance the method. So, the question is open for further research. It still may be possible that the steps towards greater power of program transformers could be made by just writing out the next formula of metasystem transition, without developing any new programs. If this does not work that simply, it should be possible to make improvements on each metasystem level. In particular, it may be possible to upgrade $\mathtt{Int}_2$ by using the knowledge that it will be always applied to the fixed function $\mathtt{Int}_1$, not to an arbitrary F, as in the case of $\mathtt{Int}_1$.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press

Acknowledgment

I greatly appreciate the discussion of the method presented here during the seminars of the Refal group in Moscow in June, 1992, especially the remarks by And. Klimov, Ark. Klimov, S. Abramov, A. Romanenko, S. Romanenko, V. Topunov. Special thanks are due to R. Glück and And. Klimov who read the manuscript and gave valuable advice on its improvement.

References

Bondorf, Anders. (1980) Automatic autoprojection of higher order recursive equations, *Lect. Notes in Comp. Sci., ESOP 90* **432**, Springer, pp.70-87.

Bowen, K. and Kowalski, R. Amalgamating language and metalanguage in logic programming. *Logic Programming*, Clark and Tarnlund (ed.) Academic Press pp.153-172.

Boyer, R.S. and Moore, J.S. (1975) Proving theorems about Lisp funcitons, *J. of ACM* **22**: pp.129-144.

Bundy, A. and Welham, B. (1981) Using metalevel inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence* vol.**16**, pp.189-212.

Burstall,R.M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J.ACM* **24** pp.44-67.

Glück, R. (1991) Towards multiple self-application, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Yale University), ACM Press: pp. 309–320.

Futamura, Y. (1971) Partial evaluation of computation process – an approach to compiler compiler, *Systems, Computers, Controls* **2**: pp.45–50.

Jones N., Sestoft P., Sondergaard H. (1985) An Experiment in Partial Evaluation: The Generation of a Compiler Generator. Jouannaud J.-P. (Ed.) *Rewriting Techniques and Applications*, Dijon, France, Lect.Notes in Comp. Sci. **202**, Springer.

Jones N., Sestoft P., Sondergaard H. (1989) MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation* **2**: pp.9–50.

Partsch, H. and Steinbrueggen, R. (1983) Program transformation systems. *ACM Comput. Surv.* **15**: pp.199–236.

Proietti, M. and Pettorossi, A. (1990) Synthesis of eureka predicates for developing logic programs. *Lect. Notes in Comp. Sci., ESOP 90* **432**: pp.306–325.

Proietti, M. and Pettorossi, A. (1991) Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Lecture Notes in Comp. Sci. PLILP 91* **528** Springer: pp.347-358.

Romanenko, A. (1991) Inversion and Metacomputation. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale University, USA pp. 12–22.

Romanenko, S.A. (1988) A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. *Partial Evaluation and Mixed Computation* Elsevier Sci. Publ.: pp.445-463.

Safra, S. and Shapiro, E. (1986) Metainterpreters for real. *Information Processing 86*, H.-J.Kugler (ed.) IFIP Congress, Dublin, Ireland. North-Holland: pp.271-278.

Sestoft, P., Sondergaard, H. (1988) A bibliography on partial evaluation, *SIGPLAN Notices*, **23** No.2: pp.19–27.

Takeuchi, A. and Furukawa, K. (1986) Partial evaluation of Prolog programs and its application to metaprogramming. *Information Processing 86*, H.-J.Kugler (ed.) IFIP Congress, Dublin, Ireland. North-Holland: pp.415-420.

Turchin, V. F. (1977) *The Phenomenon of Science*, Columbia University Press, New York.

Turchin, V. F. (1980) *The Language Refal, the Theory of Compilation and Metasystem Analysis*, Courant Computer Science Report **20**, New York University.

Turchin, V. F. (1986) The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems*, **8**: pp.292–325.

Turchin V. F. (1989) *Refal-5, Programming Guide and Reference Manual*, New England Publishing Co., Holyoke MA.

Weise, D., Conybear, R., Ruf, E., Seligman, S. (1991) Automatic online program specialization. *5th Intern. Conf. on Functional Progr. Languages and Computer Architecture. Lecture Notes in Comp. Sc.* **523** Springer, pp.165-191.

https://doi.org/10.1017/S0956796800000757 Published online by Cambridge University Press