Automating Changes of Data Type in Functional Programs *

Julian Richardson †

Abstract

In this paper I present an automatic technique for transforming a program by changing the data types in that program to ones which are more appropriate for the task. Programs are synthesised by proving modified synthesis theorems in the proofs-as-programs paradigm. The transformation can be verified in the logic of type theory.

Transformations are motivated by the presence of subexpressions in the synthesised program. A library of data type changes is maintained, indexed by the motivating subexpressions. These transformations are extended from the motivating expressions to cover as much of the program as possible.

I describe the general pattern of the revised synthesis proof, and show how such a proof can be guided by difference matching followed by rippling.

1 Introduction

In [Kowalski 79], Kowalski states that "Alteration of data structures is a way of improving algorithms by replacing an inefficient data structure by a more effective one. In a large, complex program, the demands for information made on the data structures are often fully determined only in the final stages of the program design."

In this paper I develop heuristics for applying a data-oriented program transformation technique, and present a system which implements these ideas. The system

*This research was supported by EPSRC studentship 91308481, computing resources from EPSRC GR/J80702 and grant BC/DAAD ARC Project no. 438 from the British Council. A shorter version of this paper is to appear in the Proceedings of the 10th Knowledge-Based Software Engineering Conference, Boston, USA, 12th-15th November 1995.

†Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN; Email:julianr@aisb.ed.ac.uk

constructs a proof plan (§3.3) for synthesising a transformed version of the original program.

Heuristics, encoded as methods (§3.3.2), are presented for guiding the transformation. The technique can be divided into two steps.

1. Determine which type change to make and where in the program to make it. Type changes are selected from a library in order to make certain operations on the type more efficient (§2). They are then extended to cover as much of the program as possible (§5). This is illustrated in §7 by applying a list to difference list transformation to the flatten function.

2. Synthesise new functions. Care must be taken to avoid synthesising inefficient functions. In (§3.9) I identify a proof shape which produces efficient functions and describe a proof plan which constructs such proofs. The synthesis of addition on binary numbers is used to illustrate this method in §9.

The major problem in any program transformation system is controlling the vast search space of possible transformations. Except in the case of very limited transformations, some kind of guidance is needed. Guidance can be provided by:

1. user input in an interactive system, for example asking the user to choose between several alternatives, or

2. heuristics incorporated into the system.

It is important not to swamp the user with questions, so even when the user supplies the guidance, heuristics may be necessary to limit the occasions when it is required.

Proof planning provides a framework for the expression of sophisticated heuristics. The domains of proof planning (construction of proofs, in particular inductive proofs) and program transformation (the manipulation of programs, in particular recursive programs) are closely related, and in this paper I outline how I have exploited existing proof planning techniques to automate the program transformation process.

2 Outline of technique

Commonly, the most convenient data types are used when writing a program or a theorem. In many cases, however, there are better representations which will make the program more efficient, or lead to a "better" proof of the theorem. For example, using difference lists instead of lists in a Prolog program can yield dramatic speed improvements [Hansson & Tärnlund 82], but care is needed to ensure

that the transformation is correct [Marriott & Sondergaard 88]. The change of data type requires a change of the program, replacing functions on lists (`hd`, `tl`, `append` etc.) by their equivalents on difference lists. The main effect of this transformation is that it improves the efficiency of append. We can exploit this fact by using the presence of append in the program to be transformed to motivate the change from lists to difference lists.

I generalise the reasoning above by storing profitable type changes in a library. Each entry indicates the functions which are improved by the transformation.

If we can determine that a function $f' : t' \to t'$ is more efficient than a function $f : t \to t$, then in order to replace f by f' in the program we must relate the types and functions in question. We do this requiring a retrieve function, $\rho : t' \to t$, which maps each element of t' to the element of t it represents. If ρ has an inverse ρ^{-1}, then throughout the program we can replace $f(x)$ by $\rho(f'(\rho^{-1}(x)))$. This process is formalised in §6.

3 The proof system

3.1 Introduction

An essential requirement of any program transformation system is that the transformations it applies must be provably correct, i.e. the transformed programs are equivalent to the originals. Correctness can only be determined if the programming language has a clear semantics. The logic of Martin-Löf's Constructive Type Theory [Martin-Löf 79] (§3.2) is well-suited to the task because it provides a logical system in which programs and proofs are intimately linked. This logic is implemented in the *Oyster* theorem prover [Horn & Smaill 90], a variant of NuPRL.

The proof rules of type theory are quite low-level. The *CLAM* proof planner [Bundy et al 91] constructs proof plans at a higher level of abstraction than the basic inference rules of type theory. These proof plans can subsequently be translated into object-level proofs.

3.2 Type theory

Since *Oyster*'s logic is constructive, a conjecture such as (1) can be seen as a program specification, and from its proof a function $f(x)$ satisfying (2) can be extracted. Each inference rule of the logic has a corresponding rule of program construction. The extracted functional programs are total and terminating.

$$\forall x : t \; \exists y : u \; . \; spec(x, y) \tag{1}$$

$$\forall x : t \; spec(x, f(x)) \tag{2}$$

An important point to note is that induction in the proof corresponds to recursion in the synthesised program. Thus the rippling heuristic (§3.5) which has been developed for inductive proof is of direct relevance to program synthesis.

The correspondence between proofs and programs allows program transformation to be formulated as proof transformation [Madden 91]. This permits more powerful transformations because the source and target program can satisfy the original specification without being equivalent. In addition, we have the full power of the theorem prover available to us, and transformation can be interleaved with other synthesis methods. In particular, the techniques of proof planning can be exploited.

Oyster is a goal-directed theorem prover. Rules of inference of the logic are combined using tactics. A tactic is a procedure which constructs part of the object-level proof tree.

3.3 Proof planning

3.3.1 Introduction

Machine-assisted proofs are usually made up of many small steps. A very large search space must be explored in order to generate them automatically, and they are large and hard to understand.

Proof planning [Bundy 91] addresses these issues by providing a more high-level view of the proof.

3.3.2 Methods

A method is a partial specification of a tactic. It has several slots:

Input	Sequent to which the method applies
Preconditions	Conditions which must hold for the method to apply.
Postconditions	Conditions which hold after the method has applied.
Outputs	Subgoals generated. This is a list of sequents.
Tactic	Tactic which constructs the piece of the proof corresponding to this method.

When attempting to apply a method to a sequent, firstly the sequent is unified with the input slot. If this succeeds, the preconditions are evaluated. If they

succeed, the postconditions are evaluated, leading to instantiation of the output slot.

Methods differ from tactics in that they operate on meta-level sequents, which may contain meta-variables and annotated terms (see §3.5 below).

3.3.3 The *CLAM* planner

The planner, *CLAM*, constructs a proof plan by chaining together methods, with the initial input being the proposition to be proved. If application of a method results in one or more subgoals, the planner is recursively called using each of these sequents in turn as inputs. A plan is successfully constructed when there are no remaining subgoals. *CLAM* will backtrack in an attempt to find a plan. A procedure for proving the object level theorem is constructed by replacing each method in the tree by its associated tactic.

3.4 Inductive proof

CLAM has been used extensively to mechanise proofs by induction. As an example, consider structural induction over the natural numbers:

$$\frac{P(0) \qquad\qquad P(n) \vdash P(s(n))}{\forall n\, P(n)} \tag{3}$$

When this rule of inference is applied in a proof it generates two subgoals: a base case to prove that the proposition is true for 0, and a step case to prove that the proposition is true for $s(n)$ if it is true for n.

The proof plan for induction proceeds in the step case by trying to rewrite the induction conclusion until it matches the induction hypothesis. The following sections describe how this rewriting process is guided.

3.5 Wave terms

A wave term is a term which contains wave fronts. A wave front is represented by a box immediately surrounding a term which strictly contains an underlined subterm.

These are wave terms: $\boxed{f(\underline{x})}$, $g(\boxed{g(\underline{x},\underline{x})})$. These are not: $\boxed{f(x)}$, $f(\boxed{\ })$, $\underline{f(x)}$.

3.6 Wave rules

When the `induction` method applies rule (3), the step case is annotated to indicate the differences between the induction hypothesis ($P(n)$) and the induction

conclusion ($P(\boxed{s(\underline{n})})$).

Rewrite rules of the theory are annotated using the same notation. Annotated rewrite rules are called wave rules. The ripple method [Bundy et al 93] only applies a wave rule to a subterm of a formula if the subterm unifies with the left hand side of the wave rule, and the annotations also match.

We define the skeleton of an annotated term to be the term formed by deleting all the structure which is inside a box, but not immediately inside a hole. For example, the skeleton of $p(\boxed{s(\underline{n})})$ is $p(n)$. The skeletons of the left hand side and right hand side of a wave rule must be equal, so the skeleton of a goal is unchanged when a wave rule is applied. In particular, during an inductive proof, the skeleton of the induction conclusion is equal to the induction hypothesis.

When an equation is loaded into CL^AM's internal database, an attempt is made to parse it as a wave rule. This may result in the addition of one or more records to the wave rule database.

3.7 The induction strategy

After induction and annotation of the goal, the `step_case` method applies wave rules to move the annotations further and further out in the term. Eventually, either the goal contains a copy of the induction hypothesis, in which case the induction hypothesis can be used as a rewrite rule (in a process called fertilisation), or rippling becomes stuck. Either way rippling terminates.

As an example, equations (29, 30, 31, 32) demonstrate the progressive rippling out of wave fronts until fertilisation is performed.

The rippling technique has several benefits:

1. It restricts rewriting so that termination of the rewriting process is guaranteed and search is reduced.

2. It has explanatory power, making proofs involving rewriting easier to understand.

3. The applicability of other methods can be restricted to take into account the ability of rippling to complete the subsequent proof.

3.8 Directional wave fronts

In the presentation above, wave rules move differences out through the term structure. There are occasions when we want to move differences in through the term structure, so we modify the annotations on both the wave rules and wave terms to specify a direction, either outwards:

$$plus(\boxed{s(\underline{x})}^{\uparrow}, y) \Rightarrow \boxed{s(\underline{plus(x, y)})}^{\uparrow}$$

...or inwards:

$$\boxed{s(\underline{plus(x, y)})}^{\downarrow} \Rightarrow plus(\boxed{s(\underline{x})}^{\downarrow}, y)$$

To maintain termination of rippling, we allow outward bound wave fronts to become inward bound, but not vice versa.

3.9 Difference matching

Rippling has been used extensively to guide inductive proofs. In the step case of an induction, the differences between the hypothesis and the conclusion are marked using wave fronts and rippled away by wave rules until the induction hypothesis and induction conclusion match and fertilisation can be used to complete the proof.

Difference matching [Basin & Walsh 92] extends rippling to noninductive proofs. It is a way of comparing two formulae, and annotating one such that its skeleton is equal to the other. Again, the aim is subsequently to ripple away the differences between the two formulae until they match.

3.10 Abstract data types

An important theme in modern programming languages is modularity. An abstract data type (ADT) achieves modularity by packaging a data type with its properties and the functions which access it.

In [Lowry 89], data type development is modelled by abstraction followed by implementation.

We can consider one ADT, t', to be an implementation of another, t, if there exists a retrieve function ρ from the sorts of t' to the sorts of t which obeys (9) and such that every function f in the signature of t has an analogue f' in t' and all the equations specified in the ADT for t are provable in t'. Abstraction is the inverse of implementation.

Following [Basin & Constable 93] ADTs are formulated in the type theory as Σ-types (existential types). ADTs expressed in such a way are parsed by the method `split_implementation/1` to extract function declarations, equations, wave rules and induction schemes.

4 Notation

In order to describe type changes succinctly and accurately it is necessary to introduce some notation.

A type change is determined at a point in a synthesis proof by analysing the specification at that point. Currently, however, type changes are only made in synthesis goals which correspond to object-level sequents of a restricted form:

$$Hyps \ \vdash \ \forall \bar{x} : \bar{t_x} \, \exists z : t_z \, . \, z = f(\bar{x}) \ in \ t_z$$

$Hyps$ is a list of hypotheses. $\bar{x} : \bar{t_x}$ is a set of variables with their respective types, and f is a term which contains no logical connectives. The equality can also be the other way round ($f(\bar{x}) = z$). The names of variables, types and functions may be different to those above.

To simplify the notation in this chapter, I write the function expression (the $f(\bar{x})$ above) as a shorthand for the entire specification.

A term:

$$f(x : t) : t_0 \rightsquigarrow f'(x' : t') : t'_0$$

denotes a type change in the argument of f, from type t to type t', and a change in the type of the output of $f(x)$ from type t_0 to t'_0.

If the conversion functions are $\rho : t' \rightarrow t$ and $\rho_0 : t'_0 \rightarrow t_0$, then the type change term above gives rise to the following specification of f':

$$\forall x' : t' \, . \, \rho_0(f'(x')) = f(\rho(x))$$

5 Propagation of transformations

Although f' may be more efficient than f, the overhead introduced by the insertion of retrieve functions means that the composite $\rho f' \rho^{-1}$ may not be. The simple replacement of f by $\rho f' \rho^{-1}$ is a very local transformation.

Consider a recursively defined function, with recursive case:

$$f(c(x)) = g(f(x))$$

If we make a type change of x in $g(x)$, then this recursive definition will be replaced by:

$$f(c(x)) = \rho(g'(\rho^{-1}(f(x)))) \tag{4}$$

where $g'(x')$ is the version of $g(x)$ on the new type.

Now, we can see what happens when this recursive function is called by symbolically iterating the application of f:

$$f(c(c(x))) = \rho(g'(\rho^{-1}(\rho(g'(\rho^{-1}(f(x)))))))$$

This is clearly unacceptable. If we convert x to the new type outside of the recursion, then we can avoid any conversion between recursive calls. Propagation rules specify how type changes which occur in a synthesis proof (for example a type change in g in the definition of f above) can be replaced by a single type change earlier in the proof (for example in f above). Propagation rules exploit certain information (for example the rippling in the step case below) from the original synthesis proof, and guarantee that parts of the target proof will succeed.

For example, the following rule specifies how a type change can be propagated past an induction on an unparameterised type. The name of the rule is written by the side of the rule.

$$\frac{\begin{array}{l}\text{Step case } f(\boxed{c(\underline{x})}^{\uparrow}) \Rightarrow \boxed{g(\underline{f(x)})}^{\uparrow} \\ g(x:t_0):t_0 \rightsquigarrow g'(x':t_0'):t_0'\end{array}}{f(x:t):t_0 \rightsquigarrow f'(x:t):t_0'}\text{Ind} \tag{5}$$

As described in §2, type changes are initially motivated by occurrence of certain motivating expressions in the program/synthesis proof. When a motivating expression is encountered during the synthesis, it triggers a motivation rule. I write these in the following way:

$$\frac{}{f(x:t):t_1 \rightsquigarrow f'(x':t'):t_1'}\text{motiv(f)}$$

The `typechange/2` method applies the propagation rules in a goal-directed (i.e. bottom-to-top) direction. When a specification matches the bottom part of the rule, an attempt is made to satisfy the top part of the rule. This usually involves looking ahead in the synthesis proof. In the example rule (5) above, this involves applying induction and rippling to determine the form of the step case. Eventually, either the lookahead terminates without choosing a type change, or one or more motivating expressions are encountered and the type change they specify is propagated back through the lookahead process as specified by the propagation rules, leading the a type change in the specification.

The rules which specify propagation past induction produce the most dramatic efficiency improvements. The ParmInd propagation rule, (8) specifies how a type change is propagated past induction on a parameterised type. The Feval and

Fcomp rules (6, 7) specify how a type change is propagated past function definitions. The application of a propagation rule not only causes the synthesis of a more efficient function, but also provides guarantees that parts of the target synthesis proof can be constructed.

$$\frac{\begin{array}{l} f(x) := h(x) \\ h(x:t):t_0 \rightsquigarrow h'(x':t'):t_0' \end{array}}{f(x:t):t_0 \rightsquigarrow f'(x':t'):t_0'}\text{Feval} \tag{6}$$

$$\frac{\begin{array}{l} f(x) := g(h(x)) \\ h(x:t_x):t_h \rightsquigarrow h'(x':t_x'):t_h' \\ g(x:t_h):t_g \rightsquigarrow g'(x':t_h'):t_g' \end{array}}{f(x:t_x):t_g \rightsquigarrow f'(x':t_x'):t_g'}\text{Fcomp} \tag{7}$$

$$\frac{\begin{array}{l} \text{Step case } \; f(\boxed{c(h,\underline{x})}^{\uparrow}) \Rightarrow \boxed{g(h,\underline{f(x)})}^{\uparrow} \\ g(h:t_1,x:t_0):t_0 \rightsquigarrow g'(h':t_1',x':t_0'):t_0' \end{array}}{f(x:t(t_1)):t_0 \rightsquigarrow f'(x':t(t_1')):t_0'}\text{ParmInd} \tag{8}$$

6 Correctness of transformations

For any function $f:t \to t_0$ we can indirectly synthesise an analogue $f':t' \to t_0'$ as the existential witness in a proof of:

$$\vdash \forall x:t' \; \exists z:t_0' \,.\, \rho_0(z) = f(\rho(x))$$

The original specification for f:

$$\vdash \forall x:t \, \exists z:t_0 \,.\, spec(x,z)$$

is transformed into the justification step (10).

$$\forall x:t \, \exists x':t' \,.\, \rho(x') = x \tag{9}$$

$$\forall z:t_0 \, \exists z':t_0' \,.\, \rho_0(z') = z$$

$$\vdash \forall x':t' \,.\, spec(\rho(x'), \rho_0(f'(x'))) \tag{10}$$

We can prove in the type theory that this entails the original specification as long as for each ρ we have a lemma (9) which implicitly gives an inverse.

In general if we are transforming a function of type $t_1 \to t_2 \to ... \to t_n$ to one of type $t_1' \to t_2' \to ... \to t_n'$ we will require n retrieve functions $\rho_i : t_i' \to t_i$.

7 An example transformation: lists to difference lists

A well known example of program improvement by type transformation is the use of difference lists instead of lists in a Prolog program to improve the execution of `append` (*app*), which becomes simply unification instead of a recursively defined function.

Difference lists are specified using an ADT. This incorporates a proper equality theory, which avoids the problems identified in [Marriott & Sondergaard 88], of a naïve translation of Prolog programs from lists to difference lists.

7.1 Example: transforming flatten

The example chosen is adapted from the flatten example in [Sterling & Shapiro 86, pp. 239–247], where the function transformed flattens nested lists. Instead of using nested lists I use binary trees, with leaves labelled with elements of nat.

Figure 1 shows the original program, and the transformed program, both in functional style.

The statement of the synthesis theorem contains declarations of ADTs for booleans, trees and difference lists, complete with their respective equations, induction schemes etc..

The body of the synthesis theorem is the goal:

$$\vdash \forall t : tree\, \exists l : nat\ list\, .\, l = flattent(t) \tag{11}$$

7.2 The type change

The determination of the type change is made the `typechange/2` method, which automatically chooses a transformation and decides where in the proof to apply it.

In this example, $flattent$ is defined recursively, so the method applies rule (5) to goal (11) and looks ahead in the proof by performing induction on $t : tree$ to get the step case:

$$\vdash \exists l : nat\ list\, .\, l = flattent(\boxed{node(\underline{l}, \underline{r})}^{\uparrow})$$

After rippling this becomes:

$$\vdash \exists l : nat\ list\, .\, l = \boxed{app(\underline{flattent(l)}, \underline{flattent(r)})}^{\uparrow}$$

$$flatten(node(l,r)) \;:=\; app(flatten(l), flatten(r))$$
$$flatten(leaf(n)) \;:=\; n :: nil$$

$$flatten'(t) \;:=\; ld(flatten_dl(t))$$
$$flatten_dl(node(l,r)) \;:=$$
$$\quad app_dl(flatten_dl(l), flatten_dl(r))$$

$$flatten_dl(leaf(n)) \;:=\; dl(n :: nil)$$

Figure 1: The original flatten function, and the transformed function `flatten'`. `dl` and `ld` convert between lists and difference lists.

Now the type change is motivated by the presence of $app(_,_)$. Since *app* can be improved by changing its arguments from lists to difference lists, this suggests changing the type of $flattent$ from $tree \rightarrow nat\ list$ to $tree \rightarrow dlist$ as specified by (5). This induces the synthesis of a new flatten function of type $tree \rightarrow dlist$.

The proof plan for this transformation is automatically generated by $CLAM$ and produces the transformed function, $flatten'$, in figure 1.

8 Guiding the synthesis proof

8.1 Introduction

Program synthesis in general is a difficult task, and the choice of which terms to introduce for the existential witnesses leads to a potentially infinite proof search space.

In this section, I outline the general shape of the synthesis proofs that arise after a type change, and how difference matching (§3.9) and rippling (§3.5) can be used to guide this synthesis.

The technique is illustrated using the synthesis of binary addition as an example (§9).

8.2 The proof strategy

Consider an original function specification of the form:

$$\forall x : t\ \exists z : t\ .\ z = f(x)$$

When a type change is made from type t to t' with retrieve function $\rho : t' \rightarrow t$ we get a new function specification:

$$\forall x' : t' \, \exists z' : t' \, . \, \rho(z') = f(\rho(x')) \tag{12}$$

Proof of this proposition may or may not involve induction. The final aim is, however, to find an explicit witness for z'. This witness will be the synthesised program. To ensure the efficiency of the new program, this witness should not contain applications of the retrieve functions ρ, ρ^{-1}.

If we can manipulate the proposition into the form:

$$\exists z' : t' \, . \, \rho(z') = \rho(expr(x')) \tag{13}$$

Then we can apply (14) as a rewrite rule in a left to right direction to obtain an explicit witness, $expr(x')$ for z'.

$$\forall x, y \, . \, \rho(x) = \rho(y) \leftarrow x = y \tag{14}$$

As a heuristic, I insist that the last step of the synthesis proof should be cancellation of the retrieve functions. Difference matching is applied to annotate the differences between the goal (12) and the target, the left hand side of (14).[1] This target plays the same role as the induction hypothesis in an inductive proof. Just as rippling successively reduces the differences between induction conclusion and induction hypothesis until the latter can be used in fertilisation, here it successively reduces the differences between the goal and the left hand side of (14) until it can be used as a rewrite rule to cancel the retrieve functions.

Their skeleton preserving and termination properties mean that wave rules are exactly the rewrite rules we wish to apply. Usually cancellation removes the retrieve functions entirely to leave an efficient explicit witness. Search in the proof is greatly reduced.

Difference matching is employed by the `standard_form` method, originally developed for summing series [Walsh et al 92], and the subsequent rippling is performed as in an inductive proof by the `step_case` method.

Two enhancements have been made to the `standard_form` method in order to get the closest match between goal and target:

1. Difference match against a variant of (14) in which the left hand side has been expanded using a wave rule for ρ. This maximises the size of the skeleton and so restricts subsequent application of wave front normalisation rules.

[1] In the implementation, the target formula is of the form $\forall x \, . \, \rho(x) = \rho(x)$ because this does not affect the annotations introduced in the goal.

2. Using one-step ripples, partially instantiate any existentially quantified variables in the goal.

All possible applications of these two enhancements are made, and the resulting difference matches are ranked to maximise the proportion of the term which is in the skeleton. The match with the highest measure is selected, and all the others are discarded.

8.3 A summary of the proof strategy

1. Possibly apply induction, ripple and weak fertilise, then
2. Difference match with the standard form $\forall x : t \,.\, \rho(x) = \rho(x)$ or a variant as described in §8.2.
3. Ripple.
4. When a match with the standard form is achieved, apply a cancellation rule $\rho(x) = \rho(y) \leftarrow x = y$.

9 A more complicated example: binary addition

The difference matching proof strategy is most useful in difficult synthesis proofs. In this section we apply it to a hard example, the synthesis of binary addition.

9.1 Theorem and definitions

The original synthesis theorem for addition on natural numbers is:

$$\forall x, y : nat \,.\, \exists z : nat \,.\, z = x + y$$

This allows us to synthesise an addition function from the recursion equations (20), (20) below.

After a type change from *nat* to *bin*, we get some justification goals, and a new synthesis goal:

$$\forall x, y : bin \; \exists z : bin \,.\, nat(z) = nat(x) + nat(y) \tag{15}$$

The type *bin* of binary numbers is defined as *bool list*. Binary numbers are represented in big-endian form, i.e. with the least significant digit at the head of the list. Recent work on the verification of boolean circuits by Francisco Cantu at

the Mathematical Reasoning Group in Edinburgh has shown that the choice of binary representation (big-endian or little-endian) has a significant effect on the specification and the ease or difficulty of its proof.

In addition to the definition of + we have the following wave rule for *nat*, which converts a binary number to the corresponding natural number, and a function *val* which maps binary digits (*false* or *true*) to their equivalents in *nat* (0 or s(0)):

$$nat(nil) = 0 \tag{16}$$

$$nat(\boxed{d :: \underline{x}}^{\uparrow}) \Rightarrow \boxed{val(d) + (\underline{nat(x)} + \underline{nat(x)})}^{\uparrow} \tag{17}$$

$$nat(\boxed{false :: \underline{x}}^{\uparrow}) \Rightarrow \boxed{\underline{nat(x)} + \underline{nat(x)}}^{\uparrow} \tag{18}$$

$$\forall b : bool\ b = false\ in\ bool \vee b = true\ in\ bool \tag{19}$$

$$val(false) = 0$$

$$val(true) = s(0)$$

$$x + 0 = 0$$

$$\boxed{s(\underline{x})}^{\uparrow} + y \Rightarrow \boxed{s(\underline{x + y})}^{\uparrow} \tag{20}$$

$$\boxed{s(\underline{x + y})}^{\downarrow} \Rightarrow \boxed{s(\underline{x})}^{\downarrow} + y \tag{21}$$

$$s(x) = s(y) \leftarrow x = y \tag{22}$$

$$0 + x = x$$

$$\boxed{s(s(\underline{x + y}))}^{\downarrow} \Rightarrow \boxed{s(\underline{x})}^{\downarrow} + \boxed{s(\underline{y})}^{\downarrow} \tag{23}$$

$$\boxed{a + \underline{b} + \underline{b}}^{\uparrow} + \boxed{c + \underline{d} + \underline{d}}^{\uparrow} \Rightarrow \boxed{a + c + \underline{b + d} + \underline{b + d}}^{\uparrow} \tag{24}$$

$$\boxed{s(\underline{nat(x)})}^{\downarrow} \Rightarrow nat(\boxed{inc(\underline{x})}^{\downarrow}) \tag{25}$$

$$nat(w) = nat(w) \tag{26}$$

$$val(d) + nat(w) + nat(w) = val(d) + nat(w) + nat(w) \tag{27}$$

$$nat(w) + nat(w) = nat(w) + nat(w) \tag{28}$$

Ideally, wave rules (23) and (24) would be generated automatically in a similar way to the generation of propositional wave rules [Kraan 94, pp97–100]. This would, however, introduce additional search problems.

Wave rule (25) can be extracted from a definition of the increment function. It can also be used as a specification from which the increment function can be synthesised using difference matching to guide the proof.

Equations (26,27, 28) are automatically generated when needed by the `standard_form` method. The first is the basic standard form, and the other two are standard forms obtained by rewriting (26) using wave rules (17) and (18) respectively.

9.2 The synthesis proof

This section outlines the synthesis proof which is constructed automatically by CL^AM.

The specification is (15). After performing $hd :: tl$ induction on x and y, we get:

$$\begin{array}{l} d_x : bool, x : bin \\ d_y : bool, y : bin \\ \exists z : bin \,.\, nat(z) = nat(x) + nat(y) \\ \vdash \exists z' : bin \,.\, nat(z') = nat(\boxed{d_x :: \underline{x}}^{\uparrow}) + nat(\boxed{d_y :: \underline{y}}^{\uparrow}) \end{array} \tag{29}$$

The proof proceeds as follows.

Ripple using equation (17).

$$\begin{array}{l} \exists z : bin \,.\, nat(z) = nat(x) + nat(y) \\ \vdash \exists z' : bin \,.\, nat(z') = \\ \quad \boxed{val(d_x) + (\underline{nat(x)} + \underline{nat(x)})}^{\uparrow} + \boxed{val(d_y) + (\underline{nat(y)} + \underline{nat(y)})}^{\uparrow} \end{array} \tag{30}$$

Rearrange the right hand side of the equality by rippling with wave rule (24):

$$\vdash nat(z') = \boxed{val(d_x) + val(d_y) + (\underline{nat(x) + nat(y)} + \underline{nat(x) + nat(y)})}^{\uparrow} \tag{31}$$

Weak fertilise (which removes the annotations):

$$\vdash nat(z') = val(d_x) + val(d_y) + nat(z) + nat(z) \tag{32}$$

Equation (19) specifies that *bool* is the disjoint union of a finite number of ground terms. Since there also exists an equation which can be used to evaluate $val(d_x)$, a case split on d_x is performed by the `finitetypeelim/2` method.

- $d_x = false$

  This case is easily solved by the `base_case/1` method, giving $z' = d_y :: z$.

- $d_x = true$

  The goal is simplified by `base_case/1`, and then difference matched with (27) giving:

$$\begin{array}{l} \vdash \exists v11 : bool \,.\, \exists v12 : bin \,.\, val(v11) + (nat(v12) + nat(v12)) = \\ \qquad \boxed{s(\underline{val(d_y) + (nat(z) + nat(z))})}^{\downarrow} \end{array}$$

This is rippled in with (21) to give:

$$\vdash \exists v11 : bool\,.\,\exists v12 : bin\,.\,val(v11) + (nat(v12) + nat(v12)) = \boxed{s(\underline{val(d_y)})}^{\downarrow} + (nat(z) + nat(z)) \tag{33}$$

After rippling in, a further case split is performed, this time on d_y.

- $d_y = false$

  The `guess_existential/3` method first tries $v11 = false$. Since application of the method may create a non-theorem, the planner only searches up to a predetermined fixed depth (which has been set to 7). In this case, no plan can be found. Next $v11 = true$ is tried. The goal is now:

$$\vdash \exists v16 : bool\ list\,.\,(val(true) + (nat(v16) + nat(v16))) = (s(val(false)) + (nat(z) + nat(z)))\ in\ pnat$$

  Application of the `base_case/1` method evaluates the occurrences of val, partially evaluates $+$ and applies (22) to give:

$$\exists v16 : bool\ list\,.\,(nat(v16) + nat(v16)) = (nat(z) + nat(z))\ in\ pnat$$

  This is proved by `existential_subterm/2`, giving a witness of $z' = true :: z$ for this branch of the proof.

- $d_y = true$

  The `guess_existential/3` method first tries $v11 = false$.

  The goal is:

$$\vdash \exists v16 : bool\ list\,.\,(val(false) + (nat(v16) + nat(v16))) = (s(val(true)) + (nat(z) + nat(z)))\ in\ pnat$$

  Again, `base_case/1` carries out partial evaluation to give:

$$\exists v16 : bool\ list\,.\,(nat(v16) + nat(v16)) = s((nat(z) + s(nat(z))))\ in\ pnat$$

This is difference matched by `standard_form/2` with (automatically constructed) variants of (26), as described in §8.2. The best match is achieved with (28), which gives the following annotated goal:

$$\vdash \exists v16 : bool\ list\ .\ (nat(v16) + nat(v16)) = \boxed{\underline{s((nat(z) + \boxed{\underline{s(nat(z))}}^{\downarrow}))}}^{\downarrow}\ in\ pnat$$

The `step_case/2` method ripples in with (23) and (25), giving:

$$\vdash \exists v16 : bin\ .\ nat(v16) + nat(v16) = (nat(\boxed{inc(\underline{z})}^{\downarrow}) + nat(\boxed{inc(\underline{z})}^{\downarrow}))$$

The `existential_subterm/2` method completes this branch of the proof, giving $z' = false :: inc(z)$.

Search occurs during construction of the proof plan in the (eventually failing) branch of the proof plan in which an incorrect value has been guessed for the boolean digit. Other parts of the proof plan are constructed without search.

The synthesis presented above gives the following wave rules defining binary addition (written $+_2$). There is one wave rule for each of the possible combinations of values of d_x and d_y, but these can be simplified by defining a function max on digits in the obvious way.

$$\neg(d_x = d_y = true) \rightarrow \boxed{d_x :: \underline{x}}^{\uparrow} +_2 \boxed{d_y :: \underline{y}}^{\uparrow} \Rightarrow \boxed{max(d_x, d_y) :: \underline{x +_2 y}}^{\uparrow} \quad (34)$$

$$d_x = d_y = true \rightarrow \boxed{d_x :: \underline{x}}^{\uparrow} +_2 \boxed{d_y :: \underline{y}}^{\uparrow} \Rightarrow \boxed{false :: inc(\underline{x +_2 y})}^{\uparrow} \quad (35)$$

$$inc(\boxed{false :: \underline{x}}^{\uparrow}) \Rightarrow true :: x \quad (36)$$

$$inc(\boxed{true :: \underline{x}}^{\uparrow}) \Rightarrow \boxed{false :: \underline{inc(x)}}^{\uparrow} \quad (37)$$

This is not the most efficient binary addition function. It is, however, much more efficient than unary addition. Synthesis of the standard definition of binary addition requires the introduction of a carry bit. The proof is more difficult and has not yet been automated.

10 A comparison with previous work

Choosing a data representation to improve certain program operations in used in [Schonberg et al 81]. Both [Schonberg et al 81] and [Blaine & Goldberg 91] extend an initial choice of representation to as much of the program as possible using dataflow analysis, which performs a similar role in programs to that performed by propagation rules in synthesis proofs. In both these systems, once a new data type is chosen, operations in the program are translated to ones on the new data type.

Once a type change has been determined, I automatically set up a specification for the transformed program. A fold/unfold system is used in [Darlington 80] to prove such specifications, but the proofs can involve a large amount of search. A short example from [Darlington 80] which is proved automatically by searching a fold/unfold space of 24 nodes up to depth 4 is proved without search by difference matching and rippling. A full analysis of the search improvements provided by rippling has not yet been carried out, however.

[Lowry 89] automatically derives improved ADTs, and develops an algebraic theory of data type change which allows abstraction and composition. Such a theory is also used by DTRE and has much in common with work concerning the refinement of algebraic specifications [Sannella & Tarlecki 88]. Such a theory would be of value in my system.

11 Further work

Any source synthesis proof is only used in a limited and indirect way by propagation rules. It is probable that the source proof could be exploited further than at present, possibly using analogy [Owen 90].

The type change method has been extended to allow type abstraction, but the necessary generalisation of representation functions to representation relations complicates difference matching and subsequent rippling. The difference matching proof strategy currently requires that the same type change is made on the inputs and the output of a function, and the lifting of this constraint should be investigated.

The techniques I have developed could be integrated with the more algebraic techniques of [Lowry 89, Blaine & Goldberg 91, Sannella & Tarlecki 88], and the relationship between dataflow analysis and propagation rules should be investigated.

I assume that the transformations I carry out result in more efficient programs. Complexity rules attached to proof rules such as that in [Sands 89] could be used to ensure the transformed program is more efficient than the original.

The system has been used to automate the synthesis of a number of transformations involving lists, queues, difference lists and natural numbers. Tackling some larger examples could raise interesting research problems.

12 Conclusion

In this paper I have outlined a system for transforming programs by changing the data types in the program. Data types are expressed as abstract data types, and retrieve functions specify how the original and the new types correspond in a program. The correctness of a data type change can be verified in the logic of type theory. A library specifies how certain function calls in the program can be made more efficient by certain data type changes. Propagation rules extend these changes to cover as much of the program as possible.

The synthesis theorems which arise during the transformation process are constructed with very little search by difference matching and rippling.

Acknowledgements

I would like to thank my supervisors, Professor Alan Bundy and Dr Geraint Wiggins, and the members of the Mathematical Reasoning Group for their support. I would also like to thank the anonymous referees for their comments on the KBSE version of this paper.

References

[Basin & Constable 93] David Basin and Robert Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, Logical Environments, Cambridge, 1993. Cambridge University Press.

[Basin & Walsh 92] D. Basin and T. Walsh. Difference matching. In Deepak Kapur, editor, 11th Conference on Automated Deduction, pages 295–309, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[Blaine & Goldberg 91] L. Blaine and A. Goldberg. DTRE — a semi-automatic transformation system. In B. Möller, editor, Constructing programs from specifications. North Holland, 1991.

[Bundy 91] A Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, Computational Logic: Essays in Honor of Alan Robinson, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[Bundy et al 91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Journal of Automated Reasoning, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy et al 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Artificial Intelligence, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Darlington 80] J. Darlington. The synthesis of implementations for abstract data types. Research Report DoC 80/4, Department of Computing, Imperial College of Science and Technology, 1980.

[Hansson & Tärnlund 82] A. Hansson and S-A Tärnlund. Program transformation by data structure mapping. In K. Clark and S-A Tärnlund, editors, Logic Programming, pages 117–122. Academic Press, 1982.

[Horn & Smaill 90] C. Horn and A. Smaill. Theorem proving with Oyster. Research Paper 505, Dept. of Artificial Intelligence, Edinburgh, 1990. To appear in Procs IMA Unified Computation Laboratory, Stirling.

[Kowalski 79] R. Kowalski. Algorithm = Logic + Control. Communications of ACM, 22:424–436, 1979.

[Kraan 94] I. Kraan. Proof Planning for Logic Program Synthesis. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Lowry 89] M.R. Lowry. Algorithm synthesis through problem reformulation. Unpublished PhD thesis, Stanford University, 1989.

[Madden 91] P. Madden. Automated Program Transformation Through Proof Transformation. Unpublished PhD thesis, University of Edinburgh, 1991.

[Marriott & Sondergaard 88] K. Marriott and H. Sondergaard. Prolog program transformation by introduction of difference-lists. In Proceedings International Computer Science Conference 88, pages 206–213, 1988.

[Martin-Löf 79] Per Martin-Löf. Constructive mathematics and computer programming. In 6th International Congress for Logic, Methodology and Philosophy of Science, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[Owen 90] S. Owen. Analogy for Automated Reasoning. Academic Press Ltd, 1990.

[Sands 89] D. Sands. Complexity analysis for a lazy higher-order language. In Proceedings of Glasgow Workshop on Functional Programming, Workshop Series. Springer Verlag, August 1989.

[Sannella & Tarlecki 88] D. Sannella and A. Tarlecki. Towards formal development of programs from algebraic implementations: implementations revisited. Acta Informatica, 25, 1988. Extended abstract in Proc. 12th Colloq. on Trees in Algebra and Programming, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Pisa. Springer LNCS 249, pp.96-110 (1987).

[Schonberg et al 81] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. ACM Transactions on Programming Languages and Systems, 3(2):126–143, 1981.

[Sterling & Shapiro 86] L. Sterling and E. Shapiro. The Art of Prolog. MIT Press, Cambridge, Ma., 1986.

[Walsh et al 92] T. Walsh, A. Nunes, and A. Bundy. The use of proof plans to sum series. In D. Kapur, editor, 11th Conference on Automated Deduction, pages 325–339. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.