Automated Generalisation of Function Definitions

Adam Bakewell and Colin Runciman

Department of Computer Science, University of York,
York Y010 5DD, England.
{ajb,colin}@cs.york.ac.uk

Abstract. We address the problem of finding the common generalisation of a set of Haskell function definitions so that each function can be defined by partial application of the generalisation. By analogy with unification, which derives the *most general* common specialisation of two terms, we aim to infer the *least general* common generalisation. This problem has a unique solution in a first-order setting, but not in a higher-order language. We define a smallest minimal common generalisation which is unique and consider how it might be used for automated program improvement. The same function can have many definitions; we risk over-generalisation if equality is not recognised. A normalising rewrite system is used before generalisation, so many equivalent definitions become identical. The generalisation system we describe has been implemented in Haskell.

1 Introduction

In functional programming we often use general-purpose functions from a library to make our definitions simpler and more concise. Conversely, if we recognise a common evaluation pattern in our own definitions we may wish to add a new general-purpose library function to realise that pattern. This paper is about the automatic creation of general functions. Given a set of Haskell function definitions we find their common generalisation g and a set of embedding equations which redefine each function as a partial application of g.

Example 1.

```
omnivores [] = []
omnivores (a:as) = if eatsMeat a && eatsVeg a then a:oas else oas
                   where oas = omnivores as
teens [] = []
teens (n:ns) = if 13 <= n && 19 >= n then n:tns else tns
               where tns = teens ns
```

We define `omnivores` to select animals which eat meat and vegetables from a list. It looks similar to `teens` which finds all numbers in its argument list in the range 13–19. We can define a new function `dFilter` by keeping the common

A. Middeldorp, T. Sato (Eds.): FLOPS'99, LNCS 1722, pp. 225–240, 1999.

parts of `omnivores` and `teens` and introducing new variables `p` and `q` where they differ. Now `omnivores` and `teens` can be given simpler definitions as partial applications of `dFilter`.

```
dFilter p q [] = []
dFilter p q (x:xs) = if p x && q x then x:dfxs else dfxs
                     where dfxs = dFilter p q xs
omnivores = dFilter eatsVeg eatsMeat
teens = dFilter (19 >=) (13 <=)
```

Generalisation is a routine part of programming, the original motivation for this work was to provide a tool to assist programmers. Other possible applications include selective generalisation in optimising compilers. This work raises qualitative issues like which generalisation is most useful, as well as quantitative ones such as which generalisation gives the most compact code.

We want a generalisation that is minimal in some sense. In first-order logic a *least-general generalisation* is found by *co-unification* (*anti-unification*) [13,12]. In a higher-order functional setting we can say g is less general than g' if there is a partial application of g' specialising it to g. The problem with this is g may also specialise to g', neither is less general, let alone least general.

Example 2. We could use the prelude function `filter` as the generalisation of `omnivores` and `teens`. It is less general, `filter = dFilter (const True)`; it is also more general, `dFilter p q = filter (\x -> p x && q x)`.

```
filter p [] = []
filter p (x:xs) = if p x then x:xs' else xs'
                  where xs' = filter p xs
omnivores = filter (\x -> eatsMeat x && eatsVeg x)
teens = filter (\x -> 13 <= x && 19 >= x)
```

1.1 Structure of the Paper

A Core language in which generalisations are inferred is defined in Sect. 2. This restricted syntax makes equivalences easier to identify, so there is less risk of over-generalisation. Further, even within the core language, distinct definitions can be recognised as equivalent by normalisation, Sect. 3 explains the rewrite system used for this purpose and its properties. In Sect. 4 the idea of a smallest minimal common generalisation is developed: Its inference rules are given and its properties are discussed. In Sect. 5 the generalisations produced are evaluated and potential applications for automated generalisation are considered. Related work and extensions to generalisation are discussed in Sect. 6.

2 The Core Language

Core is a subset of Haskell used for generalisation to reduce the number of ways a function can be written. It is similar to Haskell Kernel [5] but lambda or let abstractions are not allowed in arbitrary positions and features to simplify generalisation are included.

$d \in Def$	$::=$	$f\ x_a \cdots x_1 = e$ where $\mathcal{L}$	function definition
$\mathcal{L} \in Locs$	$::=$	$\{f_{i,j}\ y_1 \cdots y_a = e\}$	local function definitions
$e \in Exp$	$::=$	$k \mid v$	constructor, variable
		$\mid e\ e$	application
		$\mid$ case e *of* $\{p_i \to e_i\}[; c_j \to e]$	case expression
$u, v \in Var$	$::=$	$f_i \mid f_{i,j}$	constant, local function
		$\mid r \mid y_i$	recursive call, local function parameter
		$\mid x_i \mid c_i$	function parameter, pattern variable
$p \in Pat$	$::=$	$k\ c_1 \cdots c_a$	constructor pattern

Fig. 1. Core language grammar

2.1 Core Language Definition

Core is defined in Fig. 1. Function definitions include *local definition sets*. This allows partial applications, local recursion and higher-order substitutions to be written, whilst simplifying local definition naming and positioning.

Case expressions with simple patterns are included, as in Haskell Kernel. They include a set of constructor-pattern alternatives possibly followed by a default alternative with a variable pattern. This simplifies matters because the ordering of constructor alternatives is immaterial. To convert a standard case expression with a sequence of alternatives we remove any alternatives that can never match. For example, $(\text{case } x \text{ of } \langle 1 \to y, 1 \to z, v \to u, 2 \to w\rangle)$ becomes $(\text{case } x \text{ of } \{1 \to y\}; v \to u)$.

A sequence of zero or more expressions $(e_1 \cdots e_n)$ may be written $\overline{e}$. A set of constructor-pattern alternatives may be abbreviated A and all the alternatives of a case as AS. So $(f\ \overline{x} = \text{case } k\ \overline{e} \text{ of } \{k\ \overline{c} \to e\} \cup A)$ is a schematic function definition, the body is a case expression with constructor k applied to some expressions as its subject and a set of constructor-pattern alternatives including and whose alternatives include one with constructor k binding the variables $\overline{c}$ as its pattern, there may be any number of other alternatives.

Recursive occurrences of f_i are named r: They will become calls of the generalisation. Other free variables, including mutual recursions, are not affected by generalisation. *Reverse indexing* of function-bound variables makes adding arguments during generalisation simpler: They are added to the start of the argument list so the original can be recreated by partial application. Different kinds of bound variable are clearly distinguished. Case-bound variables and local definitions are uniquely named in a script. Function and local definition parameters are named by position.

Example 3. The Core translation of `omnivores` has case expressions instead of if-expressions and multiple equations. The variable naming and list notation are standardised. The lifted local definition requires the parameter y_1. The recursive call is replaced by r.

$$
\begin{array}{ll}
size(k) = 0 & \text{constructor} \\
size(v) = 0 & \text{variable} \\
size(e_l\ e_r) = 1 + size(e_l) + size(e_r) & \text{application} \\
size(\text{case } e \text{ of } \{a_i\}_{i=1}^n; a) = 1 + size(e) + \sum_{i=1}^{n}(1 + size(a_i)) + size(a) & \text{case} \\
size(p \to e) = 1 + size(p) + size(e) & \text{alternative} \\
size(f_{i,j}\ y_1 \cdots y_a = e)) = 1 + a + size(e) & \text{local definition} \\
size(f\ x_a \cdots x_1 = e \text{ where } \{l_i\}_{i=1}^n) = 1 + a + size(e) + \sum_{i=1}^{n}(1 + size(l_i)) & \text{definition} \\
size(\{d_i\}_{i=1}^n) = \sum_{i=1}^{n}(1 + size(d_i)) & \text{definitions}
\end{array}
$$

Fig. 2. Expression and definition sizes

$$
\begin{array}{l}
omnivores\ x_1 = \text{case } x_1 \text{ of} \\
\left\{ \begin{array}{l} [\,] \to [\,], (:)\ c_1\ c_2 \to \text{case } (\&\&)\ (eatsMeat\ c_1)\ (eatsVeg\ c_1) \text{ of} \\ \qquad\qquad\qquad\qquad \{\text{False} \to f_{1,1}\ c_2, \text{True} \to (:)\ c_1\ (f_{1,1}\ c_2)\} \end{array} \right\} \\
\text{where } \{f_{1,1}\ y_1 = r\ y_1\}
\end{array}
$$

2.2 Expression and Definition Size

To provide an abstract measure of the complexity of an expression or definition, its *size* is defined in Fig. 2 as the number of internal nodes needed to represent it as a binary tree. This is useful for comparing generalisations and deciding if generalisation is worthwhile.

2.3 Equality

Core variable naming ensures that corresponding function arguments and recursive calls are identical. Case-bound variables and local functions are uniquely named across all definitions, $e_a =_\alpha e_b$ means e_a and e_b are *equal up to renaming* of case-bound variables. We also assume that where there is no default alternative, $(\text{case } e \text{ of } A) =_\alpha (\text{case } e \text{ of } A; v \to error)$. We need to consider partial equality of case expressions so we define *equality modulo case splitting*, $(\text{case } e \text{ of } A \cup A'; v \to e') = (\text{case } e \text{ of } A; u \to \text{case } u \text{ of } A'; v \to e')$, as $e_a =_{\alpha c} e_b$. For example, $(\text{case } x_1 \text{ of } \{1 \to x_2, 2 \to x_3\}; v \to error) =_{\alpha c} (\text{case } x_1 \text{ of } \{2 \to x_3\}; u \to \text{case } u \text{ of } \{1 \to x_2\})$. Equality by definition or *extensional equality* is written $e_a = e_b$.

3 Function Normalisation

There are still many ways to define a function in Core. Showing equivalence is an undecidable problem but we can improve the chances of equivalent expressions

having the same definition – and therefore becoming part of the generalisation – by using a normalising rewrite system to eliminate differences of style.

3.1 Normalisation Rules

Normalisation makes distinct definitions identical if they are equal under a set of equivalences. We use local equivalences – no knowledge of free variables is assumed. Obvious extensions to this scheme could make use of axioms about primitive functions or established prelude function definitions.

The rules (see Appendix) are based on β,η and Case reduction [2,5], they are divided into two systems. A rule $l \rightarrow_R r$ rewrites a definition (or expression) matching l to r. Side-conditions are used to constrain the definitions that may match l.

The first normalisation system in-lines non-recursive local definitions to expose common structure, partial applications are saturated if possible or specialised (*rules* $\beta 1$–$\beta 4$). Eta-reduction (*rules* $\eta 1, \eta 2$) is applied to local definitions, it is defined inductively because an outermost application can be disguised as a case by the equivalence used in rule Case6. Case expressions are reduced to a specialisation of an alternative right-hand side if possible; if not, they are rewritten into a normal form where dead subexpressions can be identified. The merging and floating rules improve the likelihood of similar expressions being kept in the generalisation (*rules Case1–Case9*).

The second normalisation system cleans up the result of the first by removing dead local definitions, η-reducing function definitions and merging case alternatives where possible.

Example 4. The normal forms of `omnivores` and `teens` are shown here. Local definition in-lining and removal have been applied.

$$
\begin{array}{l}
\mathit{omnivores}\ x_1 = \mathrm{case}\ x_1\ \mathrm{of} \\
\left\{ \begin{array}{ll} [\,] \rightarrow [\,], (:)\ c_1\ c_2 \rightarrow & \mathrm{case}\ (\&\&)\ (\mathit{eatsMeat}\ c_1)\ (\mathit{eatsVeg}\ c_1)\ \mathrm{of} \\ & \{\mathrm{False} \rightarrow r\ c_2, \mathrm{True} \rightarrow (:)\ c_1\ (r\ c_2)\} \end{array} \right\} \\
\mathit{teens}\ x_1 = \mathrm{case}\ x_1\ \mathrm{of} \\
\left\{ \begin{array}{ll} [\,] \rightarrow [\,], (:)\ c_3\ c_4 \rightarrow & \mathrm{case}\ (\&\&)\ ((\leq)\ 13\ c_3)\ ((\geq)\ 19\ c_3)\ \mathrm{of} \\ & \{\mathrm{False} \rightarrow r\ c_4, \mathrm{True} \rightarrow (:)\ c_3\ (r\ c_4)\} \end{array} \right\}
\end{array}
$$

3.2 Properties of Normalisation

Proposition 1. *Core normalisation terminates.*

Termination is demonstrated using a well-founded ordering over the expressions. The right-hand side of each rule is lower in the ordering than the left-hand side. The ordering is monotonic – reducing a subexpression reduces its context. A simplification ordering [3] proves termination where rules remove arguments, definitions or alternatives. For the in-lining rules, replacing a (non-recursive) function by its definition is strictly reducing in the ordering, this follows from the typed lambda calculus strong normalisation property.

Proposition 2. *Core normalisation is confluent. The normal form of an expression is the same regardless of the order in which the rules are applied.*

As normalisation is terminating, it is sufficient to show *local* confluence [4]: For all critical pairs of rules (those with overlapping left-hand sides), the result of applying either rule first has a common descendant. For example, specialisation and in-lining overlap. Full in-lining gives the same result as partial specialisation followed by in-lining, once the specialisation is removed.

4 Generalisation by Co-unification

In this section we define a unique minimal generalisation of a set of Core function definitions using a form of *co-unification*. We consider why the usual method for comparing generality is inadequate, then by restricting the kind of generalisation we are willing to allow we develop an alternative definition of a minimum that meets our expectations and is equivalent to the usual minimum in the first order case. This idea is extended to full Core by defining a smallest minimal generalisation.

Definition 1. *Expression Context.*
An expression context E of arity $h \geq 0$ is an expression containing h holes written $[\cdot]$. Such a context is instantiated by applying it to a vector V of h expressions, $[e_1, \ldots, e_h]$, which fill the holes left to right. Expression vectors may be appended by writing $V \oplus V'$.

Example 5.

$$\begin{aligned}((\&\&)\ ((\leq)\ 13\ c_1)\ ((\geq)\ 19\ c_1)) &= ((\&\&)\ ([\cdot]\ c_1)\ ([\cdot]\ c_1))[(\leq)\ 13, (\geq)\ 19] \\ &= ((\&\&)\ ([\cdot]\ c_1)\ ([\cdot]\ c_1))[(\leq)\ 13] \oplus [(\geq)\ 19]\end{aligned}$$

Definition 2. *Generalisation.*
A generalisation of an indexed set of m function definitions, $\langle f_i\ \overline{x_i} = e_i \rangle_{i=1}^{m}$ (abbreviated $\langle f_i\ \overline{x_i} = e_i \rangle$) is a definition of a general function g and an indexed set of m embedding equation definitions which redefine each f_i in terms of g: $(g\ \overline{x_g} = e_g, \langle f_i = g\ \overline{e'_i} \rangle)$.

4.1 Minimal Common Generalisation (MCG)

The generality of generalisations in first-order logic is compared using the restricted instantiation preorder [1]. In the first-order case there is always a unique minimum common generalisation of a set of terms which is least in this ordering. The ordering can be adapted to a functional setting to give a generalisation ordering that places g' above g when g' can be specialised to g.

Definition 3. *Generalisation ordering.*
$(g \cdots, \langle f_i \cdots \rangle) \leq (g' \cdots, \langle f'_i \cdots \rangle) \iff \exists h \cdot (h\ \overline{x} = g'\ \overline{e}) \wedge h = g.$

Example 6. The functions $\langle f_1 = 1 + 2, f_2 = 3 + 4\rangle$ can be generalised to $(g'x = x, \langle f_1 = g'(1{+}2), f_2 = g'(3{+}4)\rangle)$ or $(gyz = y{+}z, \langle f_1 = g12, f_2 = g34\rangle)$. We can define $h\ y\ z = g'(y + z)$ and use h in place of g, therefore $g \leq g'$.

This ordering is consistent with the intuition that g' is more general than g if g' has a variable where g has some other expression. In first-order logic it can be formalised using the lattice structure of first-order terms [10], there is always a unique least generalisation which is the least upper bound of the terms. Analogously, we can define a least-general generalisation, this ought to be the one we choose to infer – intuitively it is just general enough.

Definition 4. *Least-general generalisation.*
$(g \cdots, \langle f_i \cdots\rangle)$ *is a least-general generalisation of a set of functions* F *if for all other generalisations* $(g' \cdots, \langle f'_i \cdots\rangle)$ *of* F, $(g \cdots, \langle f_i \cdots\rangle) \leq (g' \cdots, \langle f'_i \cdots\rangle)$.

In a higher-order functional setting, where we can apply the generalisation to functions, there is no unique least-general generalisation. We saw this in the introduction, `filter` and `dFilter` are instances of each other so neither is a unique minimum. We develop a minimal common generalisation (MCG) for a first-order subset of Core where there are no local functions. Generalisations are restricted so that the specialised generalisation must be *identical* to the original definitions – not merely equal. A unique minimum is defined using the idea that all parts common to the original definitions should be kept by the generalisation (and therefore not substituted by the embedding equations). We also need to rule out generalisations that include unneccesary arguments.

Definition 5. *Simple generalisation.*
$(g\ \overline{x} = E[x_1, \ldots, x_n], \langle f_i = g\ e_{1_i} \cdots e_{n_i}\rangle)$ *is a simple generalisation of the definitions* $\langle f_i\ \overline{x_i} = e_i\rangle$ *if* $\forall i \cdot E[e_{1_i}, \ldots, e_{n_i}] =_{\alpha c} e_i$.

Definition 6. *Common part.*
The common part of a set of expressions, $cp\ \langle e_i\rangle$ *is an expression context and a set of substitutions,* $(E, \langle V_i\rangle)$ *such that* $e_i =_{\alpha c} EV_i$. *It is defined in Fig. 3. The expressions have nothing in common if their common part is* $([\cdot], \langle [e_i]\rangle)$.

When all the expressions are equal up to bound-variable renaming they are the common part (1). When they are all applications 2), the common part is formed as the application of the left common part to the right common part. When they are all case expressions and there are some common constructor patterns (3), the common part is a case expression with the common part of the subject expressions as its subject and the common part of the alternatives as its alternatives. In any other case the expressions have nothing in common and a single hole is returned (4).

The common part of a set of case alternatives is found by cp'. If all the alternatives include a common constructor pattern then a new alternative is formed by finding the common part of the corresponding right-hand sides. A substitution ϕ is needed to make the corresponding pattern-bound variables

$$cp\ \langle e_i\rangle = (e, \langle[]\rangle), \text{if } \forall i \cdot e =_\alpha e_i \tag{1}$$
$$cp\ \langle e_{l_i}\ e_{r_i}\rangle = (E_l\ E_r, \langle V_{l_i} \oplus V_{r_i}\rangle) \tag{2}$$
$$cp\ \langle \text{case } e_{s_i} \text{ of } AS_i\rangle = (\text{case } E_s \text{ of } AS, \langle V_{s_i} \oplus V_{AS_i}\rangle), \text{if } AS \neq (\emptyset; v \to [\cdot]) \tag{3}$$
$$cp\ \langle e_i\rangle = ([\cdot], \langle[e_i]\rangle), \text{otherwise} \tag{4}$$

$$\text{where } (E_l, \langle V_{l_i}\rangle) = cp\ \langle e_{l_i}\rangle,\ (E_r, \langle V_{r_i}\rangle) = cp\ \langle e_{r_i}\rangle$$
$$(E_s, \langle \theta_{s_i}\rangle) = cp\ \langle e_{s_i}\rangle,\ (AS, \langle V_{AS_i}\rangle) = cp'\ \langle AS_i\rangle$$

$$cp'\ \langle A_i; v_i \to e_i\rangle$$
$$= \begin{cases} (\{p \to E_p\} \cup A; v \to E, \langle V_{p_i} \oplus V_{A_i}\rangle), \text{if } \forall i \cdot \exists (p_i \to e_{p_i}) \in A_i \cdot p = p_i\phi \\ (\emptyset; u \to E', \langle V_{e_i}\rangle), \text{ otherwise} \end{cases}$$

$$\text{where } (E_p, \langle V_{p_i}\rangle) = cp\ \langle e_{p_i}\phi\rangle$$
$$(A; v \to E, \langle V_{A_i}\rangle) = cp'\ \langle A_i - \{p_i \to e_{p_i}\}; v_i \to e_i\rangle$$
$$(E', \langle V_{e_i}\rangle) = \begin{cases} cp\ \langle e_i[u/v_i]\rangle, \text{if } \forall i \cdot A_i = \emptyset \\ ([\cdot], \langle rhs_i\rangle), \text{ otherwise} \end{cases}$$
$$rhs_i = \begin{cases} e_i[u/v_i], \text{if } A_i = \emptyset \\ \text{case } u \text{ of } A_i; v_i \to e_i, \text{otherwise} \end{cases}$$

Fig. 3. Common part of a set of expressions

identical. When there are no common constructor patterns, a variable pattern alternative is returned. Its right-hand side is the common part of the expressions $\langle e_i\rangle$ if there are no constructor-pattern alternatives left, otherwise it is a hole and rhs_i is an expression built from the remaining alternatives.

Example 7. Here we find the common part of two case expressions. The common pattern 0 is kept, the common part of False and 0 is a hole. There are no more common patterns so a new variable-pattern alternative is given with a hole as its right-hand side. The first equation is reconstructed by putting a case expression containing the remaining alternative in the second hole, the second equation just puts the expression (x_2/x_1) in the second hole.

$$cp\ \langle \text{case } x_1 \text{ of } \{0 \to \text{False}, 1 \to \text{True}\}, \text{case } x_1 \text{ of } \{0 \to 0\}; c_1 \to x_2/x_1\rangle =$$
$$((\text{case } x_1 \text{ of } \{0 \to [\cdot]\}; c_2 \to [\cdot]), \langle[\text{False}, \text{case } c_2 \text{ of } \{1 \to \text{True}\}], [0, x_2/x_1]\rangle)$$

We formalise generalisation by co-unification of a set of function definitions by equilizing their arities then taking the common part of their bodies. The generalisation is formed by dropping a new variable into each hole. For the generalisation to be minimal we need to use the same variable in two holes that are always filled by the same expression for each function. We achieve this by labelling the holes, labelling repeatedly applies *merging* which gives two holes the same label if they always have the same substitution. If the set of vectors $\langle V_i\rangle$ in the common part are viewed as a matrix with a column for each function, merging eliminates any repeated rows. For example, $label((\&\&)\,([\cdot]\,[\cdot])\,([\cdot]\,[\cdot]), \langle[((\leq)13), c_1, ((\geq)19), c_1]\rangle) = ((\&\&)\ ([\cdot]_1\ [\cdot]_2)\ ([\cdot]_3\ [\cdot]_2), \langle\{((\leq)13)/[\cdot]_1, c_1/[\cdot]_2, ((\geq)19)/[\cdot]_3\}\rangle)$.

Definition 7. *Labelled Expression Context.*
A labelled expression context E of arity n contains h labelled holes, $[\cdot]_L$. It is instantiated by applying it to a hole substitution set $\{e_1/[\cdot]_1, \ldots, e_n/[\cdot]_n\}$.

Definition 8. *Labelling and Merging.*

$$label(E, \langle [e_{1_i}, \ldots, e_{h_i}] \rangle) = (E', \langle \theta_i \rangle)$$
$$\text{where } (E[[\cdot]_1, \ldots, [\cdot]_h], \langle \{e_{j_i}\}_{j=1}^{h} \rangle) \Longrightarrow_{merge} (E', \langle \theta_i \rangle)$$
$$(E, \langle \{e_{j_i}/[\cdot]_j\} \cup \theta_i \rangle) \longrightarrow_{merge} (E\{[\cdot]_k/[\cdot]_j\}, \langle \theta_i \rangle), \text{if } \forall i \cdot e_{j_i} =_\alpha \theta_i([\cdot]_k)$$

Definition 9. *Minimum common generalisation (*MCG*).*
The simple generalisation $(g\ \overline{x} = E\{x_j/[\cdot]_j\}_{j=1}^{n}, \langle f_i = g\ e_{1_i} \cdots e_{n_i} \rangle)$ is an MCG *of $\langle f_i\ \overline{x_i} = e_i \rangle$ iff $label(cp\ \langle e_i \rangle) = (E, \langle \{e_{j_i}\}_{j=1}^{n} \rangle)$. This implies $\forall j \cdot cp\ \langle e_{j_i} \rangle = ([\cdot], \langle [e_{j_i}] \rangle)$.*

Proposition 3. *The* MCG *is the least-general simple generalisation.*

In the first-order case – where we can substitute expressions but not functions – the arity of a simple generalisation can be reduced by merging or the expressions substituted for the same variable by the embedding equations have a common part iff there is a less general simple generalisation.

4.2 Smallest MCG of Higher-Order Functions

Extending the MCG definition to full Core – where local definitions may have parameters (necessary for substituting expressions that include variables) – allows us to compare generalisations that are indistinguishable under the generalisation ordering. For example, `filter` is not an MCG of `omnivores` and `teens` because the embedding equations have a common part – both apply `&&` – `dFilter` is an MCG . An MCG is not unique. We may need to substitute functions so the holes will be filled with expressions of the form $(x_j\ \overline{v_j})$ and the embedding equations will substitute a function of the variables $\overline{v_j}$ for x_j. Because the parameters to x_j may be reordered or added to without compromising the minimality of the context we define a *smallest* MCG – the generalisation we choose to infer.

Definition 10. *Smallest minimum common generalisation (*SMCG *).*
The SMCG *of $\langle f_i\ \overline{x_i} = e_i$ where $\mathcal{L}_i \rangle$ is the smallest least-general generalisation of the form $(g\ \overline{x} = E\{x_j\ \overline{v_j}/[\cdot]_j\}_{j=1}^{n}, \langle f_i = g\ f_{1_i} \cdots f_{n_i}$ where $\{f_{j_i}\ \overline{v_j} = e_{j_i}\}_{j=1}^{n} \rangle)$ where $\forall i \cdot (E\{e_{j_i}/[\cdot]_j\}_{j=1}^{n} =_{\alpha c} e_i)$.*

Proposition 4. *The* SMCG *is unique up to variable renaming.*

$$smcg\langle f_i\ x_{a_i} \cdots x_1 = e_i \text{ where } \mathcal{L}_i\rangle = (g\ x_{a+n} \cdots x_1 = e_g[g\ x_{a+n} \cdots x_{a+1}/r], \quad (5)$$
$$\langle f_i = (r\ \bot_a \cdots \bot_{a_i+1} \text{ where } \{f_{i,j}\ \overline{v_j} = e_{j_i}\}_{j=1}^{n})[g\ f_{i,n} \cdots f_{i,1}/r]\rangle)$$

where

$$a = \max_i(a_i), (e'_i \text{ where } \mathcal{L}'_i) = (e_i \text{ where } \mathcal{L}_i)[r\ x_a \cdots x_{a_i+1}/r] \quad (6)$$

$$(E, \langle\theta_i\rangle) = label(cp\ \langle e'_i\rangle) \quad (7)$$

$$(E \text{ where } \cup_i \mathcal{L}_i, \langle\theta_i\rangle, \emptyset) \Longrightarrow_{cpb} (E' \text{ where } \mathcal{L}, \langle\{e_{j_i}\}_{j=1}^{n}\rangle, \phi) \quad (8)$$

$$\overline{v_j} = sort(\cup_i(x_k, y_k, c_k, f_{k,l} \in FV(e_{j_i}))) \quad (9)$$

$$e_g = (E' \text{ where } \mathcal{L})\{x_{a+j}\ \overline{v_j}/[\cdot]_j\}_{j=1}^{n} \quad (10)$$

$$(E \text{ where } \mathcal{L}, \langle\{f_{i,j_i}/[\cdot]_k\} \cup \theta_i\rangle, \phi)$$
$$\longrightarrow_{cpb} \begin{cases} ((E \text{ where } \mathcal{L})[f/[\cdot]_k], \langle\theta_i\rangle, \phi), \text{if } (f/\langle f_{i,j_i}\rangle) \in \phi \\ (E'', \langle\theta''_i\rangle, \phi \cup \{f_{new}/\langle f_{i,j_i}\rangle\}), \text{otherwise} \end{cases}$$

where $\{f_{i,j_i}\ y_1 \cdots y_a = e_i\} \subseteq \mathcal{L}, (E', \langle\theta'_i\rangle) = label(cp\ \langle e_i\rangle)$
$(E \text{ where } \mathcal{L} \cup \{f_{new}\ y_1 \cdots y_a = E'\}, \langle\theta_i \cup \theta'_i\rangle) \Longrightarrow_{merge} (E'', \langle\theta''_i\rangle)$

Fig. 4. SMCG inference rules

The SMCG is uniquely determined up to the ordering and naming of new variables x_j and the ordering of their parameters $\overline{v_j}$.

SMCG inference (Fig. 4) is defined in terms of the common part and redundancy removal. First any recursive calls are modified to assumes all the functions have arity a – the maximum of their arities (6). Then the common part of the defining equations is found (7), the labelled common part of the bodies is found by *cpb* (8) which creates a new local definition from any local definitions with the same arity which are called at the same position. This labelled context is turned into the generalisation body (10) by putting a new variable applied to any variables needed by the corresponding substitutions (9). Any recursive calls in g must include these new variables; the embedding equations substitute a function for each new variable, recursive calls become calls of g (5).

Example 8. The SMCG of normalised *teens* and *omnivores* is g. It is a version of `dFilter` with the local definition in-lined.

$$g\ x_3\ x_2\ x_1 = \text{case } x_1 \text{ of}$$
$$\left\{ \begin{array}{l} [\,] \to [\,], (:)\ c_5\ c_6 \to \text{case } (\&\&)\ (x_2\ c_5)\ (x_3\ c_5) \text{ of} \\ \qquad\qquad\qquad\qquad \{\text{False} \to g\ x_3\ x_2\ c_6, \text{True} \to (:)\ c_5\ (g\ x_3\ x_2\ c_6)\} \end{array} \right\}$$
$$omnivores = g\ eatsVeg\ eatsMeat,\ \ teens = g\ ((\geq)\ 19)\ ((\leq)\ 13)$$

4.3 Properties of SMCG Inference

Proposition 5. SMCG *inference is sound (up to type inference). The embedding equations are exact replacements for the original functions.*

By induction on the *cp* rules, the instantiated context yields the original expression. The treatment of recursive calls in the *smcg* rule ensures the generalisation process is sound. Merging may cause type errors though. The system is not currently safe in this respect, hole merging only when type-safe is left as further work.

Proposition 6. SMCG *inference terminates.*

The presence of local definitions means that the generalisation will not necessarily be smaller than the original definitions as in first-order logic [12]. Without local definitions the complexity of the rules is quadratic in the size of the input, owing to the union and sorting operations needed. Where local definition calls are in the common part, new local definitions are formed from their definitions. Termination is demonstrated by induction on the number of local definition calls in the common part.

5 Evaluation

The generalisation system is capable of *generating new library functions* from examples, we have already seen an example of this with `dFilter`. We are also able to generate well-known prelude functions from instances, e.g. `foldr` from list summation and multiplication functions. The SMCG might not always be the generalisation a programmer would like though, even with some normalisation, co-unification is susceptible to differences in style.

Example 9.

```
omnivores [] = []
omnivores (a:as) = if eatsMeat a && eatsVeg a then a:oas else oas
                   where oas = omnivores as
teens [] = []
teens (n:ns) = if n >= 13 && 19 >= n then n:tns else tns
               where tns = teens ns
```

The SMCG of these definitions is different to `dFilter` because `a` and `n` occur at different positions, it seems more difficult to use than `dFilter` did.

```
g q f h [] = []
g q f h (x:xs) = if (h x (f x) && q x)
                 then x:g q f h xs else g q f h xs
omnivores = g eatsVeg id (const eatsMeat)
teens = g ((>=) 19) (const 13) (>=)
```

Automated generalisation is a *reliable* way of creating new functions, but programmers may be less interested in strictly minimal functions like `g` in the previous example and prefer generalisations defined in terms of functions they are familiar with. Indeed, as programs are usually written assuming certain primitive and library functions, the generalisations should be much better if the generaliser

could work from the same background knowledge. A generalisation system incorporating more heuristics and rewriting its output in terms of prelude functions may be more appealing in this application.

If using an instance of a general-purpose function produces smaller compiled code then automatic generalisation can be used for *code storage-space optimisation.* As a compiler optimisation, our worries about the legibility of the output would not be an issue. Certainly introducing new library functions can make programs more compact. For example, the abstract size of `omnivores` and `teens` is 44, the size of their SMCG and the embedding equations is 37. Another set of tests we did looked at finding generalisations of groups of standard prelude function definitions.

Example 10.

```
sinh x = (exp x - exp (-x)) / 2; cosh x = (exp x + exp (-x)) / 2
```

These definitions are taken from the Haskell 1.4 standard prelude. Their abstract size is 20. The size of their SMCG (below) is 17. In a simple application compiled with the Glasgow Haskell Compiler on our machine, the object code size using the original definitions was 332 bytes larger than when the generalised definitions were used. With full optimisation on, the saving was 4,888 bytes. Note that a simple common subexpression analysis could not save any space in this example.

```
g x2 x1 = x2 (exp x1) (exp (-x1)) / 2
sinh = g (-); cosh = g (+)
```

The ad-hoc construction of existing libraries suggests they may contain opportunities for space and structural improvement by generalisation, our tests reinforce this view. Alternatively, libraries could be *tailored* for particular applications. The difficulty in large programs is choosing which functions to generalise. The module system may be helpful in that functions working over the same data type are likely to have something in common. Functions with similar types are also likely to have some common content.

If an interactive editor could *recognise instances* of existing library functions (by generalisation the instance with suitable functions to find one that allows the instance to be defined as a simple embedding), it could give useful hints to trainee programmers. *Finding a library function* by this method would be more reliable than simply using the polymorphic type, as in [14]. Again, we have the problem of selecting which library functions to try.

6 Related Work and Further Work

Generalisation in first-order logic was investigated by Reynolds and Plotkin [13,12] who defined a unique *least-general generalisation* of any two terms, obtained by *co-unification.* Generalisation is the basis for *program synthesis* [16] where rules are inferred from examples. Hagiya [7] uses higher-order and semantic

unification to synthesize programs by generalising first-order terms. A *higher-order unification* algorithm is given by Huet [8] which finds a higher-order unifier whenever one exists. A common generalisation always exists, SMCG inference provides a meaningful way to choose a generalisation.

There are many possible improvements to the generalisation system. Recognising more equivalences and using additional rewriting to suit the application have been mentioned. A normal form for mutually recursive local definitions would also be useful; *partial evaluation* [] could be used to specialise such definitions, the call graph could be normalised by selective inlining.

Type safety of generalisation should be investigated so we can guarantee that the type of an embedding equation is the same as the type of the function it replaces. Another potential problem with the current system is that normalisation can change the complexity of definitions. Some other useful improvements are outlines below, they all have the drawback of not giving a unique minimal generalisation.

Repeating generalisation the MCG definition could be interpreted to mean there should be as few new variables as possible. The SMCG is a first approximation to this true minimum. We only replace two new variables by one if one is redundant. In general, one variable will suffice where their substitutions can be co-unified. Repeating co-unification will give a higher approximation to the minimum. In an untyped language we can always find a generalisation that only needs one new variable, in the typed case we are more restricted.

Example 11.

```
mean x y = (x + y) / 2; avg x y = (y + x) / 2

g f h x y = (h x y + f x y) / 2
mean = g (\x y -> x) (\x y -> y); avg = g (\x y -> y) (\x y -> x)
```

By co-unifying the expressions substituted for `x3` and `x4` in the SMCG of `mean` and `avg` we get the generalisation below which needs only one new variable.

```
g f x y = (f x y + f y x) / 2
mean = g (\x y -> x); avg = g (\x y -> y)
```

Generalisation modulo theory E could give better generalisations. In the previous example we could recognise that `mean` *is* `avg` modulo commutativity. Equality modulo theory E cannot always be shown by normalisation, so the generalisation rules need extending, though where possible normalisation appears to be a better solution. Unification modulo E has been studied for many theories. Stickel's algorithm for associative-commutative functions is a good example [15]. Generalisation relative to a background theory for atomic formulas has also been studied [11]. Baader [1] defines E-generalisation in the same framework as E-unification.

Promoting embedding equations to be more than passive suppliers of substitutions could give smaller generalisations. Yet another way to generalise `mean` and `avg` is to swap the arguments and define `mean x y = avg y x`.

Unused parameters (for which embedding equations substitute $\perp$) could be used to substitute expressions. This idea is used by Furtado [6] where the co-unification of the terms $\langle x, t\rangle$ is $(x, \langle\{x/x\}, \{t/x\}\rangle)$.

Generalisation is a powerful transformation. Using ideas about program improvement, such as size reduction, seems a suitable way to utilise some of its untapped potential in a controlled way.

References

1. F Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. *Lecture Notes in Computer Science*, 488:86–97, 1991. 230, 237
2. H P Barendregt. *The Lambda Calculus: its syntax and semantics*. Elsevier Science Publishers BV, 1984. 229
3. N Dershowitz. Termination of rewriting. *J. Sym. Comp.*, 3(1-2):69–116, 1987. 229
4. N Dershowitz and J P Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 243–320. Elsevier, 1990. 230
5. J Peterson et al. *Haskell 1.4: A Non-Strict, Purely Functional Language*, 1997. 226, 229
6. A I Furtado. Analogy by generalization — and the quest for the grail. *ACM SIGPLAN Notices*, 27(1):105–113, January 1992. 238
7. M Hagiya. Synthesis of rewrite programs by higher-order unification and semantic unification. *New Generation Computing*, 8(4):403–420, 1991. 236
8. G P Huet. A unification algorithm for typed $\overline{\lambda}$-calculus. *Theor. Comp. Sci.*, 1:27–57, 1975. 237
9. N D Jones, C K Gomard, and P Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
10. K Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989. 231
11. C D Page and A M Frisch. *Inductive Logic Programming*, chapter 2: Generalization and Learnability: A Study of Constrained Atoms. London Academic Press, 1992. 237
12. G D Plotkin. A note on inductive generalization. In B Meltzer and D Mitchie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, 1969. 226, 235, 236
13. J C Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B Meltzer and D Mitchie, editors, *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, 1970. 226, 236
14. C Runciman and I Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, 1991. 236
15. M E Stickel. A unification algorithm for associative-commutative functions. *JACM*, 28(3):423–434, July 1981. 237
16. P D Summers. A methodology for lisp program construction from examples. *JACM*, 24(1):161–175, 1977. 236

Appendix: Normalisation Rules

Normalisation system 1 rules based on β, η-reduction
$reachable(e) = fix\ (\lambda F.F \cup \{f' \mid f \in F, f\ \overline{y} = e', f' \in FV(e')\})\ (FV(e))$
Assume $f'\ y'_1 \cdots y'_{a'} = e'$ where $f' \notin reachable(e')$ for rules $\beta 1$ to $\beta 4$.
$\beta 1$(**saturated application inlining**)
$\left(f'\ e_1 \cdots e_l\right) \longrightarrow_{\beta 1} \left(e'[e_1/y'_1, \ldots, e'_a/y'_{a'}]\ e_{(a'+1)} \cdots e_l\right)$, if $l \geq a'$
$\beta 2$(**arity-raising base case**)
$\left(f\ \overline{v} = f'\ \overline{e}\right) \longrightarrow_{\beta 2} \left(f\ \overline{v}\ \overline{u} = e\right)$, if $l < a' \wedge \left(f'\ \overline{e}\ \overline{u}\right) \longrightarrow_{\beta 1} \left(e\right)$
$\beta 3$(**arity-raising induction step**)
$\left(f\ \overline{v} = \text{case}\ e\ \text{of}\ \{p_i \rightarrow e_i\}; c \rightarrow e_0\right) \longrightarrow_{\beta 3} \left(f\ \overline{v}\ \overline{u} = \text{case}\ e\ \text{of}\ \{p_i \rightarrow e'_i\}; c \rightarrow e'_0\right)$
if $\exists j \cdot \left(f\ \overline{v} = e_j\right) \longrightarrow_{\beta 2, \beta 3} \left(f\ \overline{v}\ \overline{u} = e'_j\right) \wedge k \neq j \iff (e'_k = e_k\ \overline{u})$
$\beta 4$(**partial application specialisation**)
$\left(f\ \overline{v} = E[f'\ e_1 \cdots e_l]\right) \longrightarrow_{\beta 4} \left(f\ \overline{v} = E[f''\ u_1 \cdots u_k]\right)$, if $l < a'$
where $\langle u_1, \ldots, u_k \rangle = sort(y_i, c_i \in \text{FV}(\{e_1, \ldots, e_l\}))$
$f''\ u_1 \cdots u_k\ y_{l+1} \cdots y_a = e'[e_1/y_1, \ldots, e_l/y_l]$
$\eta 1$(**eta reduction base case**)
$\left(f\ \overline{u}\ v = e\ v\right) \longrightarrow_{\eta 1} \left(f\ \overline{u} = e\right)$, if $v \notin FV(e)$
$\eta 2$(**eta reduction induction step**)
$\left(f\ \overline{u}\ v = \text{case}\ e\ \text{of}\ \{p_i \rightarrow e_i\}; c \rightarrow e_0\right) \longrightarrow_{\eta 2} \left(f\ \overline{u} = \text{case}\ e\ \text{of}\ \{p_i \rightarrow e'_i\}; c \rightarrow e'_0\right)$
if $v \notin FV(e) \wedge \forall j \cdot \left(f\ \overline{u}\ v = e_j\right) \longrightarrow_{\eta 1, \eta 2} \left(f\ \overline{u} = e'_j\right)$

Normalisation system 1 rules based on case reduction
Case1(**variable pattern case reduction**)
$\left(\text{case}\ e_s\ \text{of}\ \emptyset; v \rightarrow e\right) \longrightarrow_{Case1} \left(e[e_s/v]\right)$
Case2(**case variable specialisation**)
$\left(\text{case}\ e_s\ \text{of}\ A; v \rightarrow e\right) \longrightarrow_{Case2} \left(\text{case}\ e_s\ \text{of}\ A; v \rightarrow e[e_s/v]\right)$
Case3(**case merging**)
$\left(\text{case}\ e_s\ \text{of}\ A; v \rightarrow \text{case}\ e_s\ \text{of}\ \{k_i\ \overline{c_i} \rightarrow e_i\}; u \rightarrow e\right)$
$\longrightarrow_{Case3} \left((\text{case}\ e_s\ \text{of}\ A \cup \{k_i\ \overline{c_i} \rightarrow e_i \mid k_i\ \overline{c'_i} \rightarrow e'_i \notin A\}; u \rightarrow e)[e_s/v]\right)$
Case4(**pattern matching case reduction**)
$\left(\text{case}\ k\ \overline{e}\ \text{of}\ \{k_i\ \overline{c_i} \rightarrow e_i\}; v \rightarrow e\right) \longrightarrow_{Case4} \begin{cases} e_j[\overline{e}/\overline{c_j}], & \text{if } k = k_j \\ e[k\ \overline{e}/v], & \text{if } \forall i, k \neq k_i \end{cases}$
Case5(**floating case out of case**)
$\left(\text{case}\ (\text{case}\ e_s\ \text{of}\ \{p_i \rightarrow e_i\}; v \rightarrow e)\ \text{of}\ AS\right)$
$\longrightarrow_{Case5} \left(\text{case}\ e_s\ \text{of}\ \{p_i \rightarrow \text{case}\ e_i\ \text{of}\ AS\}; v \rightarrow \text{case}\ e\ \text{of}\ AS\right)$
Case6(**floating apply into case**)
$\left((\text{case}\ e_s\ \text{of}\ \{p_i \rightarrow e_i\}; v \rightarrow e)\ \overline{e}\right) \longrightarrow_{Case6} \left(\text{case}\ e_s\ \text{of}\ \{p_i \rightarrow e_i\ \overline{e}\}; v \rightarrow e\ \overline{e}\right)$
Case7(**dead alternative elimination**)
$\left(\text{case}\ e_s\ \text{of}\ \{k\ \overline{v} \rightarrow e\} \cup A; v \rightarrow E\ [\text{case}\ e'_s\ \text{of}\ \{k\ \overline{u} \rightarrow e''\} \cup A'; a]\right)$
$\longrightarrow_{Case7} \left(\text{case}\ e_s\ \text{of}\ \{k\ \overline{v} \rightarrow e\} \cup A; v \rightarrow E\ [\text{case}\ e'_s\ \text{of}\ A'; a]\right)$, if $e_s =_\alpha e'_s$
Case8(**repeated computation elimination**)
$\left(\text{case}\ e_s\ \text{of}\ \{p \rightarrow e\} \cup A; a\right) \longrightarrow_{Case8} \left(\text{case}\ e_s\ \text{of}\ \{p \rightarrow e[p/e_s]\} \cup A; a\right)$

Case9(**variable pattern simplification**)
$\big(\,\text{case}\, e_s \,\text{of}\, A; v \to e\,\big) \longrightarrow_{Case9} \begin{cases} \text{case}\, e_s \,\text{of}\, A \cup \{k\,\overline{c} \to e[k\,\overline{c}/v]\}, \text{ if } U = \{k\} \\ \text{case}\, e_s \,\text{of}\, A, \text{ if } U = \emptyset \end{cases}$
where $U = \{\text{constructors } k \text{ of type } \tau(e_s) \mid (k\,\overline{c} \to e) \notin A\}$

Normalisation system 2 rules
$\beta 5$(**local definition elimination**)
$\big(\,e \,\text{where}\, \mathcal{L}\,\big) \longrightarrow_{\beta 5} \big(\,e \,\text{where}\, \{(f \cdots) \in \mathcal{L} \mid f \in \mathit{reachable}(e)\}\,\big)$
Case10(**redundant alternative elimination**)
$\big(\,\text{case}\, e_s \,\text{of}\, \{p \to e\} \cup A; v \to e'\,\big)$
$\longrightarrow_{Case10} \begin{cases} \text{case}\, e_s \,\text{of}\, A; v \to e', \text{ if } e =_\alpha e' \wedge A \neq \emptyset \\ \text{case}\, e_s \,\text{of}\, \{p_0 \to e\}; v \to e', \text{ if } e =_\alpha e' \wedge A = \emptyset \end{cases}$
where p_0 is the least constructor of type $\tau(e_s)$