PROGRAM SEPARATION AS A BASIS FOR DEFINITIONAL HIGHER ORDER PROGRAMMING

GÖRAN FALKMAN

ABSTRACT. We describe a program separation scheme based on a notion of form and content of an algorithm. We show that this type of program separation is one way of making algorithms an explicit part of declarative programming. We also show how this type of program separation can be used as a basis for definitional higher order programming.

1. Introduction

The basis for all modern declarative programming languages like ML, Haskell, and Prolog is that the programmer should be more concerned with *what* the program should do and less with *how* the program should accomplish some task.

The actual algorithms are often rather *implicit* in such languages since they are hidden under layers of very high-level notions like pattern-matching, unification, and backtracking. So if you want to study algorithms in this context there must be a way of making algorithms a more *explicit* part of declarative programming without losing to much of the declarative reading of the programs.

Just as higher order functions and meta-programming have been subjects to intensive studies in the functional programming and logic programming community respectively, we are in the area of definitional programming interested in the notions of *higher order definitions* and *definitional higher order programming.*

In this paper we describe a program separation scheme based on a notion of *form* and *content* of an algorithm. We show that this type of program

---

Key words and phrases. Programming methodology, definitional programming, program separation.

This work was supported by the Swedish Research Council for Engineering Sciences (TFR) and the Swedish National Board for Industrial and Technical Development (NUTEK).

It is part of the work done within the ESPRIT working group GENTZEN

separation is one way of making algorithms an explicit part of declarative programming. We also show how this type of program separation can be used as a basis for definitional higher order programming.

The work presented here is based on some earlier work on program separation [5].

As an introductory example, consider the recursive definition of addition:

$$\mathcal{P} \quad \begin{cases} plus(0,m) = m, \\ plus(s(n),m) = s(plus(n,m)). \end{cases}$$

The intended algorithm associated with this definition is given implicitly in proving that $plus(n,m) = n+m$ with respect to $\mathcal{P}$. By induction: If $n=0$, then the definiens of $plus(n,m)$, $\mathcal{P}(plus(n,m))$, is m and thus in this case $plus(n,m) = n+m$. So assume $plus(n,m) = n+m$ with respect to $\mathcal{P}$. Now $\mathcal{P}(plus(s(n),m)) = s(plus(n,m))$, so by the induction hypothesis $plus(s(n),m) = s(n)+m$. The algorithm may be formulated as follows:

(1) if $n=0$, then take the definiens of $plus(n,m)$ in $\mathcal{P}$, $\mathcal{P}(plus(n,m))$, as the result of the computation of $plus(n,m)$,
(2) if $n=s(k)$, then the result of the computation of $plus(n,m)$ is given by computing $plus(k,m)$ and then applying s to the result.

In this algorithm we really only use primitive operations on definitions; looking up the definiens of an expression and checking whether a given expression is defined or not. So we may reformulate the algorithm in the following manner:

$$Plus(x) = \begin{cases} \mathcal{P}_1(x) & \text{if } \mathcal{P}_1(x) \cap \mathrm{Dom}(\mathcal{P}_1) = \emptyset, \\ \mathcal{P}_2(Plus(\mathcal{P}_1(x))) & \text{if } \mathcal{P}_1(x) \cap \mathrm{Dom}(\mathcal{P}_1) \neq \emptyset. \end{cases}$$

where $\mathcal{P}_1$ and $\mathcal{P}_2$ are the following definitions:

$$\mathcal{P}_1 \quad \begin{cases} plus(0,m) = m, \\ plus(s(n),m) = plus(n,m). \end{cases}$$

$$\mathcal{P}_2 \quad \{ n = s(n)$$

$\mathcal{P}_1$ and $\mathcal{P}_2$ is the canonical separation of $\mathcal{P}$ in order to find definitions such that addition is locally definable, i.e., to isolate the local operations needed to compute additions on the basis of $\mathcal{P}$. *Plus* then describes the global structure of the algorithm.

It is natural to think of this type of program separation as a separation of *form* and *content*. *Plus* gives the recursive form of the algorithm and $\mathcal{P}_1$ and $\mathcal{P}_2$ give the particular content needed to compute addition.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to the target language. In Section 3 we introduce a notion of form and content of an algorithm by studying a couple of examples.

2. Programming with definitions

The target language of this paper is based on an existing definitional programming language called GCLA[1] [1, 2, 3, 4, 7].

Compared to GCLA we use an idealized syntax and, what is more important, we have extended the language by allowing multiple object-level definitions and more general guards and provisos. We use the same operational semantics as in GCLA.

2.1. Basic building blocks. The basic building blocks of the language are *variables*, *constants*, *terms*, *atoms*, and *conditions*.

2.1.1. *Variables.* Variables are denoted by strings with a capital initial letter, e.g., "X", "Xs", "$A\_long\_variable$" etc.

2.1.2. *Constants.* Constants are denoted by strings beginning with a lowercase letter, e.g., "x", "$aConstant$" etc.

2.1.3. *Terms.* Terms are built up using variables and constants:

(i) All variables are terms,
(ii) All constants are terms,
(iii) If f is a functor and $T_1, \ldots, T_n$ are terms, then $f(T_1, \ldots, T_n)$ is a term.

2.1.4. *Atoms.* Atoms are terms which are not variables:

(i) All constants are atoms,
(ii) If f is a functor and $T_1, \ldots, T_n$ are terms, then $f(T_1, \ldots, T_n)$ is an atom.

2.1.5. *Conditions.* Conditions are built up using, $\perp$, $\top$, and terms:

(i) $\perp$ is condition. (Falsity),
(ii) $\top$ is a condition. (Truth),
(iii) All terms are conditions,
(iv) If C_1 and C_2 are conditions, then $C_1 \rightarrow C_2$ is a condition,
(v) If C_1 and C_2 are conditions, then $C_1 \,; C_2$ is a condition,
(vi) If C_1 and C_2 are conditions, then $C_1 \,, C_2$ is a condition.

2.2. Definitions and programs. *Definitions* are built up by *definitional clauses* and using definitions we form *programs*.

2.2.1. *Definitional clauses.* If a is an atom and C is a condition, then $a = C$ is a definitional *clause*.

A is referred to as the *head* of the clause and C is referred to as the *body* of the clause.

We may also have *guarded* definitional clauses. If a is an atom, C is a condition, and $G_1, \ldots, G_n$ are *guards*, i.e., simple syntactical equality and inequality constraints on variables like "$X = Y$", "$f(Y) \neq f(g(Z))$", "$W \geqslant 1$" etc., then $a\#\{G_1 \wedge \cdots \wedge G_n\} = C$ is a guarded definitional clause.

[1]Is to be pronounced "Gisela".

2.2.2. *Definitions.* A definition $\mathcal{D}$ is an ordered set of (guarded) definitional clauses:

$$\mathcal{D} \quad \begin{cases} a = A. \\ b = B. \\ c = C. \\ d = D. \\ \quad \vdots \end{cases}$$

Definitions will be denoted by $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, ...

2.2.3. *Programs.* A *program* $\mathbf{P}$, is a set of definitions, $\mathbf{P} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ for $n > 1$.

2.3. The meaning of a program. A program $\mathbf{P}$ is a set of definitions. In the definitions we define atoms in terms of certain conditions, but these conditions are not defined in the definitions themselves. Instead we select one of the definitions in $\mathbf{P}$ to interpret all the conditions in the other definitions. This definition will be called the *rule definition* (or the meta-level definition) and it will be denoted by $\boldsymbol{\mathcal{R}}$. The definitions in $\mathbf{P} \setminus \{\boldsymbol{\mathcal{R}}\}$ are referred to as the *object-level definitions* (or simply the definitions). We continue to denote object-level definitions by $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, ...

The difference between the object-level definitions and the rule definition is that in the latter we have access to some primitive operations on definitions which are used in the communication between the rule definition and the object-level definitions.

The purpose of $\boldsymbol{\mathcal{R}}$ is to give meaning to the conditions that are used in the object-level definitions. To give this meaning we define *inference rules* that interpret all the different object-level conditions. We also define *search strategies* that guide the search among the inference rules.

When defining inference rules and search strategies we use the conditions defined in Section 2.1.5. Rather than having a "meta-meta-level definition" interpreting these conditions the meaning of the conditions used in $\boldsymbol{\mathcal{R}}$ is given by a sequent calculus, $\mathcal{DOLD}$, which is the same calculus that gives the operational semantics of GCLA [7].

Thus a program $\mathbf{P}$ consists of $\boldsymbol{\mathcal{R}}$ and at least one object-level definition, and an interpretation of $\mathbf{P}$ is obtained only by putting the object-level definitions in the context of $\boldsymbol{\mathcal{R}}$.

2.3.1. *Inference rules.* An inference rule (or simply a rule) interprets one of the object-level conditions. The general form of an inference rule is:

$$\begin{aligned} Rule(PT_1, \ldots, PT_n) = \; & P_1, \ldots, P_k, \\ & (PT_1 \rightarrow Seq_1), \ldots, (PT_n \rightarrow Seq_n) \\ & \rightarrow Seq. \end{aligned}$$

where

- $PT_1, \ldots, PT_n$ are proof terms, i.e., partially instantiated functional expressions representing the proofs of the premises, Seq_i,
- $P_1, \ldots, P_k$ are provisos, i.e., side conditions on the applicability of the rule. k may be equal to 0,
- Seq and $Seq_1, \ldots, Seq_n$ are sequents on the form $\Gamma \vdash C$ where Γ is a finite set of conditions and C is a condition. Intuitively $\Gamma \vdash C$ could be read as: "C follows from Γ."

One way to read this is: "If $P_1, \ldots, P_k$ hold and PT_1 proves Seq_1, ..., and PT_n proves Seq_n, then $Rule(PT_1, \ldots, PT_n)$ proves Seq."

2.3.2. *Search strategies.* A search strategy (or simply a strategy) guides the search among the inference rules. A strategy has the following general form:

$$Strategy = (P_1 \to Seq_1), \ldots, (P_n \to Seq_n).$$
$$Strategy = PT_1, \ldots, PT_m.$$

where

- $PT_1, \ldots, PT_m$ are proof terms,
- $P_1, \ldots, P_n$, for $n \geqslant 0$, are provisos,
- $Seq_1, \ldots, Seq_n$, for $n \geqslant 0$, are sequents.

You can read a strategy as: "If P_1 holds and some PT_i, $1 \leqslant i \leqslant m$, proves Seq_1, then $Strategy$ proves Seq_1, ..., or if P_n holds and some PT_i, $1 \leqslant i \leqslant m$, proves Seq_n, then $Strategy$ proves Seq_n."

2.3.3. *Provisos.* A proviso is a condition on the applicability of a rule or a strategy. Intuitively you can think of a proviso as a black box performing some operation that has to succeed in order for the rule or strategy to succeed. Some provisos may have side-effects like adding constraints to variables or generating substitutions.

Some of the provisos are used in the communication between $\mathcal{R}$ and the object-level definitions. These provisos are built up using a set of primitive operations on definitions which are entirely justified by the interpretation of the object-level definitions as definitions.

The most important of these primitive operations are:

- $\mathcal{D}(a)$ which gives the *definiens* of the term a in the definition $\mathcal{D}$. In short, $\mathcal{D}(a)$ is the maximal set of all bodies of clauses in $\mathcal{D}$ whose heads are unifiable with each other and with a, i.e., $\mathcal{D}(a) = \{A \mid a = A \in \mathcal{D}\}$. There can be more than one definiens for a term a.
  The reader is referred to [7] for a formal definition of $\mathcal{D}(a)$ and to [2] for details on the computational aspects of $\mathcal{D}(a)$,
- $\mathrm{Dom}(\mathcal{D})$ which gives the *domain* of a definition $\mathcal{D}$, i.e., $\mathrm{Dom}(\mathcal{D}) = \{a \mid \exists A(a = A \in \mathcal{D})\}$,
- $\mathrm{unify}(A, B)$ which is used to unify the object-level terms A and B.

Using the operations above we can for example construct a proviso for testing whether something is in the domain of a certain definition, i.e., "$a \in \mathrm{Dom}(\mathcal{D})$".

2.4. Queries. If Seq is a sequent on the form $\Gamma \vdash C$ and S is a (meta-level) condition then $S \Vdash Seq$. is a *query*.

We may interpret a query as asking for an object-level substitution σ such that $S\phi$ proves $\Gamma\sigma \vdash C\sigma$ for some meta-level substitution ϕ.

3. Program separation

3.1. A notion of form and content. In this section we are going to introduce a notion of form and content of an algorithm. We will do this by studying an example. This will lead us to a definition of an abstract recursive form of a class of very simple recursive algorithms.

3.1.1. *A motivating example.* As an example we study the canonical definition for finding the last element of a list.

Example 3.1.1. Consider the following recursive definition for finding the last element of a list:

$$\mathcal{L} \quad \begin{cases} last((x::nil)) = x, \\ last((x::xs)) = last(xs) \quad \text{if } xs \neq nil. \end{cases}$$

where nil denotes the empty list and $::$ is the cons operator.

The intended algorithm associated with this definition can be described as follows:

(1) if $l = (x::nil)$, then take the definiens of $last(l)$ in $\mathcal{L}$, $\mathcal{L}(last(l))$, as the result of the computation of $last(l)$,
(2) if $l = (x::xs)$, where $xs \neq nil$, then the result of the computation of $last(l)$ is given by computing $last(xs)$.

Now, what we want to do is to separate, so to speak, the (recursive) *form*, or the global structure, of the algorithm from the particular *content* needed to compute *last*.

To find the particular content needed to compute *last* we try to separate $\mathcal{L}$ into definitions such that the operation of finding the last element of list is *locally definable*, i.e., we try to isolate the *local operations* needed to compute *last* on the basis of $\mathcal{L}$.

Intuitively *locally defined* means that all operations needed to compute the result from an argument are present in the definitions but *global* information, like which local operation to be performed and when, are not [6].

In the case of *last*, it turns out that we do not need to separate $\mathcal{L}$ any further; whenever we reach something on the form $last((x::nil))$ in the computation of *last*, the next thing to do is to return x, and whenever we find something on the form $last((x::xs))$ the next step is to compute $last(xs)$. But this is precisely what $\mathcal{L}$ says, so *last* is already locally defined in $\mathcal{L}$.

To extract the recursive form of the algorithm we study the computation of $last((1::(2::(3::nil))))$. We see that all we have to do to compute $last((1::(2::(3::nil))))$ is to look up the definiens of $last((1::(2::(3::nil))))$ in $\mathcal{L}$ and recursively compute $\mathcal{L}(last((1::(2::(3::nil)))))$, i.e., $last((2::(3::nil)))$. The recursion stops when we find an expression that is not defined in $\mathcal{L}$, i.e., an element.

So in the computation of $last((1::(2::(3::nil))))$ we really only use primitive operations on definitions; looking up the definiens of an expression and checking whether a given expression is defined or not.

We may therefore reformulate the algorithm in the following manner:

$$Last(x) = \begin{cases} \mathcal{L}(x) & \text{if } \mathcal{L}(x) \not\subseteq \mathrm{Dom}(\mathcal{L}), \\ Last(\mathcal{L}(x)) & \text{if } \mathcal{L}(x) \subseteq \mathrm{Dom}(\mathcal{L}). \end{cases} \tag{1}$$

We may read (1) as: "If $\mathcal{L}(x)$, the definiens of x in $\mathcal{L}$, is not defined in $\mathcal{L}$ (the base case), then take $\mathcal{L}(x)$ as the result of the computation of $Last(x)$. Else if $\mathcal{L}(x)$ is defined in $\mathcal{L}$ (the recursive case), then the result of the computation of $Last(x)$ is given by recursively computing $Last(\mathcal{L}(x))$."

It is natural to think of this type of separation as a separation of form and content. We say that (1) gives the (recursive) *form* of the algorithm whereas $\mathcal{L}$ gives the particular *content* needed to compute *last*.

The natural way to implement a separated algorithm, as the one above for computing *last*, is to define the content as object-level definitions and to define the form by strategies and rules in the rule definition.

In the case of *last*, the implementation of $\mathcal{L}$ is straightforward:

$$\mathcal{D}_1 \quad \begin{cases} last([X]) = X. \\ last([X|Xs])\#\{Xs \neq []\} = last(Xs). \end{cases}$$

$\mathcal{R}$ consists of a rule with two clauses corresponding to the two cases of (1):

$$\mathcal{R} \quad \begin{cases} f(1) = & (\mathcal{D}_1(X) \not\subseteq \mathrm{Dom}(\mathcal{D}_1)), \\ & \mathrm{unify}(\mathcal{D}_1(X), F) \to (X \vdash F). \\ f(2) = & (\mathcal{D}_1(X) \subseteq \mathrm{Dom}(\mathcal{D}_1)), \\ & (f(\_) \to (\mathcal{D}_1(X) \vdash F)) \to (X \vdash F). \end{cases} \tag{2}$$

A typical query would then be:

$$f(\_) \Vdash last([1,2,3,4,5]) \vdash X.$$

which has the solution $X = 5$.

3.1.2. *An abstract recursive form.* In the definition of (1) in Example 3.1.1, we did not use any information about the particular definition at hand. Thus it should be possible to describe the recursive form of the algorithm for finding the last element of a list without referring to any specific definition.

We get the following generalization of (1):

$$F(x) = \begin{cases} \mathcal{D}(x) & \text{if } \mathcal{D}(x) \not\subseteq \mathrm{Dom}(\mathcal{D}), \\ F(\mathcal{D}(x)) & \text{if } \mathcal{D}(x) \subseteq \mathrm{Dom}(\mathcal{D}). \end{cases} \tag{3}$$

where $\mathcal{D}$ is some definition.

In contrast to (1), (3) does not describe the recursive form of a specific algorithm but the *abstract* recursive form of a *class* of algorithms.

Further examples of algorithms belonging to this class are the canonical definitions of enumerating the members of a list, enumerating the members of a binary tree, deciding whether an element is a member of a list or not, generating suffixes of a list, returning the element with index i from a list, and others.

Exactly what constitutes this class remains an open question.

We also generalize (2) in the same way:

$$\boldsymbol{\mathcal{R}} \quad \begin{cases} f(1,\mathcal{D}) = & (\mathcal{D}(X) \not\subseteq \mathrm{Dom}(\mathcal{D})), \\ & \mathrm{unify}(\mathcal{D}(X), F) \to (X \vdash F). \\ f(2,\mathcal{D}) = & (\mathcal{D}(X) \subseteq \mathrm{Dom}(\mathcal{D})), \\ & (f(\_,\mathcal{D}) \to (\mathcal{D}(X) \vdash F)) \\ & \to (X \vdash F). \end{cases} \tag{4}$$

The rule f is now parameterized with a definition and since f itself is a definition, this is a kind of *definitional* higher order programming.

To find the last element in the list $[1,2,3,4,5]$ we would now pose the query:

$$f(\_,\mathcal{D}_1) \Vdash \mathit{last}([1,2,3,4,5]) \vdash X.$$

where $\mathcal{D}_1$ is the same definition as in Example 3.1.1.

3.2. Splitting and collecting. In many algorithms, a *splitting* and *collecting* part can be discerned. In the splitting part the computation is split up into several sub-computations and in the collecting part the results from these sub-computations are collected together to form the final result.

In some of these algorithms, the result of the collecting part is the result of the whole computation while in others some additional computing is needed in order to get the final result.

In this section we are going to study the abstract recursive forms of very simple recursive algorithms which can be described in terms of a splitting and collecting part.

As we will see these forms are natural extensions of the abstract recursive form given by (3) in Section 3.1.2.

3.2.1. *Algorithms that build structures.* Algorithms that build some kind of a data structure are very common. As an example of such an algorithm we study the canonical algorithm for appending two lists.

Example 3.2.1. Consider the following recursive definition for appending two lists:

$$\mathcal{A}\quad \begin{cases} append(nil, ys) = ys, \\ append((x::xs), ys) = (x::append(xs, ys)). \end{cases}$$

The algorithm normally associated with this definition can be described as follows:

(1) if $xs = nil$, then take the definiens of $append(xs, ys)$ in $\mathcal{A}$ as the result of the computation of $append(xs, ys)$,
(2) if $xs = (z::zs)$, then the result of the computation of $append(xs, ys)$ is given by computing $append(zs, ys)$ and then applying $::$ to z and the result.

The algorithm described above is really *implicit* in $\mathcal{A}$ since there is nothing in $\mathcal{A}$ that indicates that we must first look inside $(x::append(xs, ys))$ to compute $append(xs, ys)$ and then apply $::$ to x and the result. What we like to do is to make the algorithm a more *explicit* part of $\mathcal{A}$.

We do this by applying the program separation scheme outlined in Section 3.1, i.e., we try to separate the recursive form of the algorithm from the particular content needed to compute *append*.

One way to find the particular content needed to compute *append* is to separate $\mathcal{A}$ into the following definitions:

$$\mathcal{A}_1\quad \begin{cases} append(nil, ys) = ys, \\ append((x::xs), ys) = x, \\ append((x::xs), ys) = append(xs, ys). \end{cases}$$

$$\mathcal{A}_2\quad \begin{cases} x = (x::xs) & \text{if } x \neq nil \wedge x \neq (y::ys), \\ xs = (x::xs) & \text{if } xs = nil \vee xs = (y::ys). \end{cases}$$

The intuition behind $\mathcal{A}_1$ is that whenever we reach something on the form $append(nil, ys)$ in the computation of *append* then we return ys and whenever we find something on the form $append((x::xs), ys)$ the computation is split up into two sub-computations: x and $append(xs, ys)$.

Similarly, the intuition behind $\mathcal{A}_2$ is that whenever we reach something that is not a list, i.e., an element, then we construct a list from that element and some list xs and whenever we find something that is a list then we construct a new list from that list and some element x. $\mathcal{A}_2$ can be seen as the reverse of the inductive definition of lists of a given type.

To find the recursive form of the algorithm we notice that to compute for example $append((1::(2::nil)), (3::nil))$ all we have to do is to look up the definiens of $append((1::(2::nil)),(3::nil))$ in $\mathcal{A}_1$ and continue with the subcomputations 1 and $append((2::nil),(3::nil))$. Now we should put 1 in front of a list, namely the list we get by appending $(3::nil)$ to $(2::nil)$, and we can do that by simply looking up the definiens of 1 in $\mathcal{A}_2$. When the recursive computation of $append((2::nil),(3::nil))$ eventually returns, we should put an element, i.e., 1, in front of the resulting list and we can do that by simply looking up the definiens of the resulting list in $\mathcal{A}_2$. The recursion stops when we find something that is not defined in $\mathcal{A}_1$, i.e., an element.

So in the computation of $append((1::(2::nil)),(3::nil))$ we really only use primitive operations on definitions and we may therefore formulate the algorithm as follows:

$$Append(x) = \begin{cases} \mathcal{A}_1(x) & \text{if } \mathcal{A}_1(x) \cap \mathrm{Dom}(\mathcal{A}_1) = \emptyset, \\ \mathcal{A}_2(Append(\mathcal{A}_1(x))) & \text{if } \mathcal{A}_1(x) \cap \mathrm{Dom}(\mathcal{A}_1) \neq \emptyset. \end{cases} \tag{5}$$

We say that (5) gives the *recursive form* of the algorithm whereas $\mathcal{A}_1$ and $\mathcal{A}_2$ together give the particular *content* needed to compute *append*.

$\mathcal{A}_1$ and $\mathcal{A}_2$ are implemented by two object-level definitions:

$$\mathcal{D}_3 \quad \begin{cases} append([],Ys) = Ys. \\ append([X|Xs],Ys) = X. \\ append([X|Xs],Ys) = append(Xs,Ys). \end{cases}$$

$$\mathcal{D}_4 \quad \begin{cases} X\#\{X \neq [] \wedge X \neq [Y|Ys]\} = [X|Xs]. \\ Xs\#\{Xs = [] \vee Xs = [Y|Ys]\} = [X|Xs]. \end{cases}$$

The form (5) is implemented by the following rule definition:

$$\mathcal{R} \quad \begin{cases} f(1) = (\mathcal{D}_3(X) \cap \mathrm{Dom}(\mathcal{D}_3) = \emptyset), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_3(X), F) \rightarrow (X \vdash F). \\ f(2) = (\mathcal{D}_3(X) \cap \mathrm{Dom}(\mathcal{D}_3) \neq \emptyset), \\ \qquad\quad (\{A_1, \ldots, A_n\} = \mathcal{D}_3(X)), \\ \qquad\quad (g(\_) \rightarrow (A_1 \vdash F)), \ldots, (g(\_) \rightarrow (A_n \vdash F)) \rightarrow (X \vdash F). \\ \\ g(1) = (X \notin \mathrm{Dom}(\mathcal{D}_3)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_4(X), F) \rightarrow (X \vdash F). \\ g(2) = (X \in \mathrm{Dom}(\mathcal{D}_3)), \\ \qquad\quad (f(\_) \rightarrow (X \vdash FX)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_4(FX), F) \rightarrow (X \vdash F). \end{cases} \tag{6}$$

The first clause of the rule f corresponds to the base case. If the definiens of an expression consists of more than one member we evaluate all the members to the same value. This is to ensure that the two lists created from the element and the recursive call respectively are equal. This is taken care of in the second clause of f. Thus, it is in this rule that the collecting is performed.

The two clauses of the rule g correspond to the second case in (5).

To concatenate the lists $[1,2]$ and $[3,4]$ we pose the query:

$$f(\_) \Vdash append([1,2],[3,4]) \vdash X.$$

which has the solution $X = [1,2,3,4]$.

What happens in the algorithm given by (5) above is that we use $\mathcal{D}_3$ to present values and recursive calls and then use $\mathcal{D}_4$ to build a structure based on the number of recursive calls.

3.2.2. *Computing in the collecting part.* So far we have only considered examples where the collecting part consists of building a structure on basis of a given datatype. In this section we consider a simple example where we actually compute in the collecting part.

Example 3.2.2. Consider the following (naive) recursive definition for reversing a list:

$$\mathcal{R} \quad \begin{cases} reverse(nil) = nil, \\ reverse((x::xs)) = append(reverse(xs),(x::nil)). \end{cases}$$

where *append* is the operation of appending two lists.

The intended algorithm associated with this definition can be described as follows:

(1) if $xs = nil$, then take the definiens of $reverse(xs)$ in $\mathcal{R}$, $\mathcal{R}(reverse(xs))$, as the result of the computation of $reverse(xs)$,
(2) if $xs = (y::ys)$, then the result of the computation of $reverse(xs)$ is given by computing $reverse(ys)$ and then appending $(x::nil)$ to the result.

In order to make the above algorithm a more explicit part of $\mathcal{R}$ we try to separate $\mathcal{R}$ into definitions such that *reverse* is locally definable. One way to do this is to separate $\mathcal{R}$ into the following definitions:

$$\mathcal{R}_1 \quad \begin{cases} reverse(nil) = nil, \\ reverse((x::xs)) = x, \\ reverse((x::xs)) = reverse(xs). \end{cases}$$

$$\mathcal{R}_2 \quad \begin{cases} x = append(xs,(x::nil)) & \text{if } x \neq nil \wedge x \neq (y::ys), \\ xs = append(xs,ys) & \text{if } xs = nil \vee xs = (y::ys). \end{cases}$$

We can understand $\mathcal{R}_1$ as: "If we reach something on the form $reverse(nil)$ in the computation of *reverse* then we return *nil* and if we find something on the form $reverse((x::xs))$ the computation is split up into two computations: x and $reverse(xs)$."

Similarly, we can understand $\mathcal{R}_2$ as: "If we reach something that is not a list, i.e., an element, then we construct an append structure from some list xs and the list consisting of that element, and if we find something that is a list then we construct an append structure from that list and another list ys."

It is easy to see that the recursive form of this algorithm must be very similar to (5); in both cases we use the two results from the recursive step to build a data structure. The important difference is that in the case of reversing a list the data structure is intermediate—it is not the final result of the computation—we need to do some additional computing on the structure, i.e., evaluating the append structure to a list.

We get the following recursive form of the algorithm:

$$Rev(x) = \begin{cases} \mathcal{R}_1(x) & \text{if } \mathcal{R}_1(x) \cap \mathrm{Dom}(\mathcal{R}_1) = \emptyset, \\ Append(\mathcal{R}_2(Rev(\mathcal{R}_1(x)))) & \text{if } \mathcal{R}_1(x) \cap \mathrm{Dom}(\mathcal{R}_1) \neq \emptyset. \end{cases} \tag{7}$$

where *Append* is the algorithm for appending two lists.

Perhaps to gain some additional intuition of what this program separation is all about, one may think of the content of the algorithm as the part of the algorithm that defines on *which* objects the next step of the algorithm should act upon, and the form of the algorithm as the part that defines *what* exactly is the next step of the algorithm.

If we translate this algorithm into a program, $\mathcal{R}_1$ and $\mathcal{R}_2$ are implemented by the following two object-level definitions:

$$\mathcal{D}_5 \quad \begin{cases} reverse([]) = []. \\ reverse([X|Xs]) = X. \\ reverse([X|Xs]) = reverse(Xs). \end{cases}$$

$$\mathcal{D}_6 \quad \begin{cases} X\#\{X \neq [] \wedge X \neq [Y|Ys]\} = append(Xs,[X]). \\ Xs\#\{Xs = [] \vee Xs = [Y|Ys]\} = append(Xs,Ys). \end{cases}$$

The form (7) is implemented by the following rule definition:

$$\mathcal{R} \quad \begin{cases} f(1) = (\mathcal{D}_5(X) \cap \mathrm{Dom}(\mathcal{D}_5) = \emptyset), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_5(X), F) \rightarrow (X \vdash F). \\ f(2) = (\mathcal{D}_5(X) \cap \mathrm{Dom}(\mathcal{D}_5) \neq \emptyset), \\ \qquad\quad (\{A_1, \ldots, A_n\} = \mathcal{D}_5(X)), \\ \qquad\quad (g(\_) \rightarrow (A_1 \vdash FX)), \ldots, (g(\_) \rightarrow (A_n \vdash FX)), \\ \qquad\quad (append \rightarrow (FX \vdash F)) \rightarrow (X \vdash F). \\ \\ g(1) = (X \notin \mathrm{Dom}(\mathcal{D}_5)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_6(X), F) \rightarrow (X \vdash F). \\ g(2) = (X \in \mathrm{Dom}(\mathcal{D}_5)), \\ \qquad\quad (f(\_) \rightarrow (X \vdash FX)), \\ \qquad\quad \mathrm{unify}(\mathcal{D}_6(FX), F) \rightarrow (X \vdash F). \end{cases} \tag{8}$$

where *append* is some strategy or rule defined elsewhere for appending two lists, e.g., the rule f in (6).

This rule definition is essentially the same as (6) but with the important difference that we use an external strategy (or rule) *append* in the second clause of f for evaluating the intermediate append structure built by g to a list.

If we would like to reverse the list $[1,2,3,4,5]$ we can pose a query like:

$$f(\_) \Vdash reverse([1,2,3,4,5]) \vdash X.$$

which has the solution $X = [5,4,3,2,1]$.

3.2.3. *Abstract recursive forms.* In the definitions of (5) and (7) in Example 3.2.1 and Example 3.2.2 respectively, we did not use any information about the particular definitions at hand. So it should be possible to describe the recursive form of the algorithms for computing *append* and *reverse* without referring to any specific definition.

Consider the following generalization of (5):

$$F(x) = \begin{cases} \mathcal{D}_1(x) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset, \\ \mathcal{D}_2(F(\mathcal{D}_1(x))) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset. \end{cases} \tag{9}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are some definitions.

Just like (3) in Section 3.1.2, (9) describes the abstract recursive form of a *class* of algorithms. Further examples of algorithms belonging to this class are the canonical definitions of finding the maximum and minimum of two natural numbers, computing the length of a list, substituting one element for another in a list, duplicating all the members in a list etc.

We also generalize the rule definition in Example 3.2.1 in the same way:

$$\mathcal{R} \quad \begin{cases} f(1,\mathcal{D}_1,\mathcal{D}_2) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset), \\ & \mathrm{unify}(\mathcal{D}_1(X), F) \rightarrow (X \vdash F). \\ f(2,\mathcal{D}_1,\mathcal{D}_2) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset), \\ & (\{A_1,\ldots,A_n\} = \mathcal{D}_1(X)), \\ & (g(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (A_1 \vdash F)), \\ & \ldots, \\ & (g(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (A_n \vdash F)) \\ & \rightarrow (X \vdash F). \\ & \\ g(1,\mathcal{D}_1,\mathcal{D}_2) = & (X \notin \mathrm{Dom}(\mathcal{D}_1)), \\ & \mathrm{unify}(\mathcal{D}_2(X), F) \rightarrow (X \vdash F). \\ g(2,\mathcal{D}_1,\mathcal{D}_2) = & (X \in \mathrm{Dom}(\mathcal{D}_1)), \\ & (f(\_,\mathcal{D}_1,\mathcal{D}_2) \rightarrow (X \vdash FX)), \\ & \mathrm{unify}(\mathcal{D}_2(FX), F) \rightarrow (X \vdash F). \end{cases} \tag{10}$$

The rules f and g are now a kind of higher order definitions since they are parameterized with two definitions $\mathcal{D}_1$ and $\mathcal{D}_2$.

To concatenate the two lists $[1,2]$ and $[3,4]$ we would now pose the query:

$$f(\_,\mathcal{D}_3,\mathcal{D}_4) \Vdash append([1,2],[3,4]) \vdash X.$$

where $\mathcal{D}_3$ and $\mathcal{D}_4$ are the same definitions as in Example 3.2.1.

Similarly we get the following generalization of (7):

$$F(x) = \begin{cases} \mathcal{D}_1(x) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset, \\ G(\mathcal{D}_2(F(\mathcal{D}_1(x)))) & \text{if } \mathcal{D}_1(x) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset. \end{cases} \tag{11}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are some definitions and G is some function defined elsewhere.

The form (11) describes the abstract recursive form of a class of algorithms. Further examples of algorithms belonging to this class are the canonical definitions of flattening a list, computing the factorial of a natural number, computing the fibonacci-numbers, computing the sum and product of all the members in a list of natural numbers, listing a binary tree in pre-, in-, and post-order checking if a binary tree is full etc.

We also generalize the rule definition in Example 3.2.2 in the same way:

$$\mathcal{R} \quad \begin{cases} f(1, \mathcal{D}_1, \mathcal{D}_2, PT) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) = \emptyset), \\ & \mathrm{unify}(\mathcal{D}_1(X), F) \\ & \rightarrow (X \vdash F). \\ f(2, \mathcal{D}_1, \mathcal{D}_2, PT) = & (\mathcal{D}_1(X) \cap \mathrm{Dom}(\mathcal{D}_1) \neq \emptyset), \\ & (\{A_1, \ldots, A_n\} = \mathcal{D}_1(X)), \\ & (g(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (A_1 \vdash FX)), \\ & \ldots, \\ & (g(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (A_n \vdash FX)), \\ & (PT \rightarrow (FX \vdash F)) \\ & \rightarrow (X \vdash F). \\ & \\ g(1, \mathcal{D}_1, \mathcal{D}_2, PT) = & (X \notin \mathrm{Dom}(\mathcal{D}_1)), \\ & \mathrm{unify}(\mathcal{D}_2(X), F) \\ & \rightarrow (X \vdash F). \\ g(2, \mathcal{D}_1, \mathcal{D}_2, PT) = & (X \in \mathrm{Dom}(\mathcal{D}_1)), \\ & (f(\_, \mathcal{D}_1, \mathcal{D}_2, PT) \rightarrow (X \vdash FX)), \\ & \mathrm{unify}(\mathcal{D}_2(FX), F) \\ & \rightarrow (X \vdash F). \end{cases} \tag{12}$$

Since the definitions f and g are parameterized with two definitions, $\mathcal{D}_1$ and $\mathcal{D}_2$, this is yet another example of how this type of program separation can be used as a basis for definitional higher order programming.

To reverse the list $[1, 2, 3, 4, 5]$ we pose the query:

$$f(\_, \mathcal{D}_5, \mathcal{D}_6, append) \Vdash reverse([1, 2, 3, 4, 5]) \vdash X.$$

where $\mathcal{D}_5$ and $\mathcal{D}_6$ are the same definitions as in Example 3.2.2 and *append* is some strategy or rule defined elsewhere for appending two lists, e.g., the rule f in (6).

References

1. M. Aronsson, *GCLA user's manual*, Research Report T91:21A, SICS, 1991.
2. ———, *GCLA, the design, use, and implementation of a program development system*, Ph.D. thesis, Department of Computer and Systems Sciences, University of Stockholm, 1993.
3. M. Aronsson, L-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin, *The programming language GCLA—a definitional approach to logic programming*, New Generation Computing **7** (1990), no. 4, 381–404.
4. M. Aronsson, P. Kreuger, L. Hallnäs, and L-H. Eriksson, *A survey of GCLA—a definitional approach to logic programming*, Extensions of Logic Programming, Proceedings of the 1st International Workshop held at the SNS, Universität Tübingen, 1989 (P. Shroeder-Heister, ed.), Springer Lecture Notes in Artificial Intelligence, vol. 475, Springer-Verlag, 1990.
5. G. Falkman, L. Hallnäs, and O. Torgersson, *Program separation in GCLA*, Proceedings of the Post-Conference Workshop on Proof-Theoretical Extensions of Logic Programming, Santa Margherita Ligure, Italy, 18 June 1994 (A. Momigliano and M. Ornaghi, eds.), June 1994, pp. 31–37.
6. L. Hallnäs, *WM94: program separation in GCLA*, Proceedings of La Wintermöte '94, Lundsbrunn, January 18–20, January 1994, pp. 93–94.
7. P. Kreuger, *GCLA II, a definitional approach to control*, Extensions of Logic Programming, Proceedings of the 2nd International Workshop held at SICS, Sweden, 1991 (L-H. Eriksson, L. Hallnäs, and P. Shroeder-Heister, eds.), Springer Lecture Notes in Artificial Intelligence, vol. 596, Springer-Verlag, 1992, pp. 239–297.

Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden

E-mail address: falkman@cs.chalmers.se