ISRN SICS-R--91/10--SE

A Definitional Approach to the Combination of Functional and Relational Programming

by

Martin Aronsson

SICS research report
R91:10
ISSN 0283-3638

A Definitional Approach to the Combination of Functional and Relational Programming

Martin Aronsson
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden
email: martin@sics.se

1991-06-14

Abstract
We show how the programming language GCLA can be used to naturally express both relational and functional programs in an integrated framework. We give a short introduction to GCLA, and to the theory of partial inductive definitions on which GCLA is based. GCLA is best regarded as a logic programming language, but instead of saying that the query follows from the program in some a priori given logic, we say that the program defines the logic in which the query is proved. We then demonstrate how to implement both relational and functional programs as well as a combination of them in GCLA.

1. Introduction

Traditionally, logic programming and functional programming are two separated branches of computer science. The effort spent on merging these two often tend to simply incorporate one branch into the other. For some examples see [DG-L86].
As another approach, we use the notion of *partial inductive definitions* [Hal87] to represent both relations and functions in the same framework. The theory of partial inductive definitions is a generalization of the theory of inductive definitions, to which the ability to make assumptions has been added. We give a short introduction to these ideas, as well as the language GCLA [Aro90a, Aro90b], which is based on the theory of partial inductive definitions.

We will refer to programs based on SLD resolution as *relational programs*, since the term *logic programs* refers to a larger set of languages, including GCLA. We shall, however, point out that GCLA is not a logic programming language in the traditional sense; rather it should be looked upon as a language for *expressing* the inference rules of a logic. When, for example, a Prolog system proves a query, it proves that the query follows from the program in a given logic (first order classical logic), but when a query in GCLA is proved, the GCLA system proves that there is a deduction of that query in the logic specified by the program.

2. Definitions and GCLA

Section 2.1 and 2.2 borrow from the ideas put forth in a research report [EH88] written by Lars-Henrik Eriksson and Lars Hallnäs. We will keep this description informal and short. The interested reader is referred to [HS-H88, Aro90a, Hal87] for more detailed explanations.

2.1. Inductive definitions

An inductive definition specifies the members of a set in terms of other members of the same set. A typical example of an inductive definition is

> $\emptyset$ is a list.
> For all l and x, if l is a list, then $x.l$ is a list.

Here $x.l$ is **defined** to be a list if l is a list. We can see that $a.\emptyset$ is a list, but that $a.b$ is not a list. $a.b$ is defined to be a list if b is a list. Thus if b was a list, $a.b$ would be too. However, there is nothing that defines b to be a list under any circumstances, so $a.b$ cannot be a list (it is assumed that nothing else can be a list except as defined by this definition).

We can formalize this slightly by writing

$$\emptyset \Leftarrow$$
$$x.l \Leftarrow l \qquad \text{for all } x,l$$

The arrow should be read as "defines". In other words, $\emptyset$ is trivially defined (to be a list). $x.l$ is defined (to be a list) if l is. The line

$$x.l \Leftarrow l \qquad \text{for all } x,l$$

should be regarded only as a finite description of how to generate the infinite number of expressions defining every possible list.

An expression of the form $e \Leftarrow E$ is called a *clause*.

Informally, we have said that an atom e belongs to a certain set (defined by a certain definition $\mathcal{D}$) if there is a clause of the form $(e \Leftarrow E) \in \mathcal{D}$, and all $e' \in E$ belong to that set. This suggests a family of deductive calculi, one for each $\mathcal{D}$, to deduce what atoms belong to the set defined by $\mathcal{D}$. We have the following inference rule:

$$\frac{\{\vdash e' \mid e' \in E\}}{\vdash e} \vdash\mathcal{D} \qquad \text{under the condition that } (e \Leftarrow E) \in \mathcal{D}$$

The notation $\vdash e$ should be read "e holds". The notation $\vdash F$, where F is a set, will be short for "$\vdash e$, for all $e \in F$". For brevity, we will often say that "e is derivable" when we really mean that "$\vdash e$" is derivable.

2.2. Partial inductive definitions (PID)

We will now generalize the concept of an inductive definition to that of a "partial inductive definition" [Hal87].

A clause $e \Leftarrow E$ expresses the fact that e holds under the assumption that all $e' \in E$ hold. So far we have only been permitted to **state** such facts. However, one could imagine the possibility of having a "hypothetical" reasoning similar to hypothetical reasoning in logic, where we assume that certain atoms hold and **show** such a clause to be a fact.

To do this we permit assumptions to the left of the derivability symbol ($\vdash$), modify the $\vdash \mathcal{D}$-rule to pass assumptions to the left, and add axioms of the form $e \vdash e$. We will call the terms to the left of $\vdash$ *assumptions*, and the term to the right of $\vdash$ a *conclusion.*

The $\vdash\mathcal{D}$ rule now looks like

$$\frac{\{F \vdash e' \mid e' \in E\}}{F \vdash e} \vdash\mathcal{D} \qquad \text{under the condition that } (e \Leftarrow E) \in \mathcal{D}$$

As an example, consider the definition of lists in section 2.1. $a.b.x$ holds (is a list) according to the definition of lists, under the assumption that x holds (is a list).

$$\cfrac{\cfrac{\cfrac{}{x \vdash x}}{x \vdash b.x}}{x \vdash a.b.x}$$

To turn this result into a clause, we need a new inference rule:

$$\frac{E, F \vdash e}{F \vdash E \Rightarrow e} \vdash \Rightarrow$$

We can now complete the derivation:

$$\frac{x \vdash a.b.x}{\vdash x \Rightarrow a.b.x} \vdash \Rightarrow$$

To complete our introduction to partial inductive definitions, two additional inference rules are stated, i.e. the two that are dual to $\vdash_{\mathcal{D}}$ and $\vdash_{\Rightarrow}$:

If a set of atoms $e_1, ..., e_n$ are derivable and C is derivable from e, then C should be derivable from $e_1, ..., e_n \Rightarrow e$. This is analogous to the inference rule in sequent calculus for an implication among the assumptions.

$$\frac{\{F \vdash C' \mid C' \in E\} \quad F, e \vdash C}{F, E \Rightarrow e \vdash C} \Rightarrow\vdash$$

In a sense, an application of the $\vdash_{\mathcal{D}}$ rule corresponds to going from a definition to the atom being defined, i.e. if e is defined by the clause $e \Leftarrow E$, if we know E we can derive e. We complete the extension of the calculi by adding an inference rule that permits us to go from an atom to its definition, i.e to show that C is derivable under the assumption e, where e is defined by the clause $e \Leftarrow E$, we could show that C is derivable under the assumption E. In case several different clauses could define e (if the definition was nondeterministic), we could not know which one actually did define it, consequently we would have to have one premise for each clause defining e. This is analogous to or-introduction to the left in the sequent calculus.

$$\frac{\{F, E \vdash C \mid (E \Rightarrow e) \in \mathcal{D}\}}{F, e \vdash C} \mathcal{D}\vdash$$

Note especially that under the assumption of an atom e that could not possibly hold, i.e. for which there is no clause $e \Leftarrow E$ in the inductive definition defining it, then anything holds.

As an example of a partial inductive definition, consider the definition defining truth in a model for classical logic, and compare it with the standard recursive definition:

$p \wedge q \Leftarrow p,q$	$\varphi \wedge \psi$ is true if φ is true and ψ is true
$p \vee q \Leftarrow p$	$\varphi \vee \psi$ is true if φ is true
$p \vee q \Leftarrow q$	$\varphi \vee \psi$ is true if ψ is true
$p \rightarrow q \Leftarrow (p \Rightarrow q)$	$\varphi \rightarrow \psi$ is true if: if φ is true then ψ is true
$\neg p \Leftarrow (p \Rightarrow \bot)$	$\neg\varphi$ is true if φ is not true

Note the close correspondence between the definition in the meta language and the formal definition. Also note how hypothetical statements are used to model implication and negation.

2.3. GCLA (Generalized horn Clause LAnguage)

GCLA is a programming language based on the theory of partial inductive definitions. For each rule in the theory of partial inductive definitions, there is a corresponding rule in GCLA, which together with a given search strategy forms a system for proving statements. For a complete description of GCLA, the reader is referred to [Aro90b, HS-H88].

One big difference between GCLA and partial inductive definitions is that GCLA has the notion of *variables*. A variable is essentially the same thing in GCLA and in the language

Prolog, although a variable should be considered more as a metalogical device in GCLA, since the theory of partial inductive definitions does not have variables. Informally, given a universe $\mathcal{U}$, a term in the definition containing a variable represents all terms where that variable has been substituted with a term from the universe $\mathcal{U}$.

When executing a query in GCLA, GCLA looks for substitutions for the variables in the query that proves that the query holds according to the given definition (the loaded program). Queries are on the form `Assumptions \- Conclusion`, and the system works bottom up, i.e the query is the bottommost sequent.

Consider the list definition given in section 2.1. A corresponding GCLA program is

```
list([]).
list([F|R]) <= list(R).
```

and the corresponding queries are

```
\- list([a,b,c])
```

and

```
list(x) \- list([a,b,x]).
```

The $\vdash\mathcal{D}$ rule in GCLA corresponds to SLD resolution, so an ordinary Prolog program can be seen as an inductive definition, and Prolog execution can be seen as repeatedly applying the rule $\vdash\mathcal{D}$. Thus, all (pure) Prolog programs can be executed by a GCLA system.

A *guard* in GCLA is a restriction on the quantification of variables occurring in the head of a clause. A clause `p(X) <= q(X)` with the guard `X ≠ 1` is written

```
p(X) <= [X ≠ 1],q(X).
```

Typically, guards contain a restriction that various variables should differ from various templates. The variables are constrained for the rest of the execution with the restrictions, c.f. constraint programming and domain variables [Hen89].

The inference rules can be tried in various orders, which can be determined by the user, and for certain goals various rules can be applied. More specifically, for a given goal sequent, there is a determined *search order* , determining the order in which the inference rules are to be tried. This is accomplished by a general scheme implementing a control structure, which makes it possible to guide the execution. It is beyond the scope of this paper to describe this control structure, the interested reader is referred to [Aro90b] or [Kre91]. We will describe where necessary what effect the structure impose on the execution. It should however be stressed that the general control structure we are talking about here has a clear and well defined semantics [Kre91].

A control mechanism that needs to be mentioned is the ability to restrict terms from being applicable by the $\mathcal{D}$-rules. If some term is not supposed to be used by a $\mathcal{D}$-rule (typically the result of evaluating a functional expression), we can define the term as being canonical, and not an object for further refinements. One could think of the term as being defined circular,

```
term <= term
```

i.e. when expanded the term returns itself (c.f. functional programming, where, for example, a number evaluates to itself). We will in this paper make use of this possibility and declare lists (`[...]`) and the constant nil (`[]`) as such terms.

2.4. Program example 1: A toy knowledge base

As an example, consider the definition below, which implements a small expert system (This example can also be found in [Aro90a]). Note that the definition itself does not contain any trivial rules, only rules for drawing conclusions from certain given data.

```
disease(cold) <= symptom(cough),temp(normal).
disease(pneumonia) <=
      symptom(persistent_cough),symptom(chill),temp(high).
```

To this definition, we have to declare the terms `symptom(...)` and `temp(...)` as being nonapplicable for the $\mathcal{D}$-rules. We also assume a certain search order, which gives as result that the axiom rule is tried first, then the rules to the right of $\vdash$, and lastly the rules to the left of $\vdash$.

A typical query to this definition is when we have some symptoms, and ask for a possible diagnosis, for example

```
symptom(cough), temp(normal) \- disease(X)
```

To this query, the system response is `X = cold`.

We can also pose queries, where we do not have complete knowledge of the symptoms,

```
symptom(cough), temp(Y) \- disease(X)
```

and the system answers `X = cold, Y = normal`.

As a last example, we pose a hypothetical query: If we suppose that someone has pneumonia, what follows from that assumption,

```
disease(pneumonia) \- X
```

The first (trivial) answer is `X = disease(pneumonia)`, but as we backtrack, other, more interesting answer substitutions follows: `X = symptom(persistent_cough)`, `X = symptom(chill)`, and finally, `X = temp(high)`.

3. Functional definitions

We will now consider how functional definitions can be written in GCLA. A more thorough description of functional definitions and their theoretic properties can be found in [Fre90]. We will here concentrate on how these definitions can be utilized and executed.
The description will be informal. We will describe how a functional definition is written and utilized by examples.

3.1 Program example 2: The functional definition for add

The function we will consider is ordinary addition for successor arithmetic, i.e. the natural numbers. All definable functions can be defined and executed within the GCLA framework. It is also possible to write nondeterministic functions, but we will not consider that opportunity here.

The definition for adding two natural numbers is

```
add(0,X) <= X.
add(s(Y),X) <= succ(add(Y,X)).
```

and should be read:
- To add `0` and `X` (where `X` could be instantiated to a function call) is `X` (or the evaluated value of `X`).
- To add the successor of some `Y` with `X` is defined to be the successor of adding `Y` and `X`.

This definition defines the base case and the recursive case, but lacks a clause defining `succ`, a substitution schema. The `succ`-clause must calculate the value of `add(Y,X)`, and substitute the expression `add(Y,X)` with its value.

```
succ(X) <= ((X -> Y) -> s(Y)).
```

It should be read as: if `X` can be calculated to some `Y`, we can replace `succ(X)` with `s(Y)`". It is the clause for `succ` that, in a sense, performs the actual computation. It is a substitution schema, which substitute its (unevaluated) argument with an (evaluated) value.

The terms `s(...)` and `0` are canonical values, that is, they cannot be further evaluated, and therefore they are not applicable to the $\mathcal{D}$-rules. One could think of these terms as defined

```
s(X) <= s(X)
0 <= 0.
```

i.e. the terms are evaluated to themselves (c.f. traditional functional languages), as described earlier.

With the clauses for `add` and `succ`, we are now in a position to compute values from `add`-expressions.

The goals are on the form *`expression \- value`*, and during the execution *`expression`* is stepwise calculated to the value *`value`*. For example, the goal

```
add(0,0) \- X
```

computes `add(0,0)` to `0` in one step, which is then unified with the variable `X`.

3.2 Eager evaluation

With eager evaluation, we try to simplify the goal expression as much as possible before returning a value. This means that we try to compute a canonical value from the given expression. We omit the rule $\vdash\mathcal{D}$, since it will not contribute anything to the computation of the expressions.

To get eager evaluation, we define a search order, which states that the inference rules should be tried in the following order:

$$\mathcal{D}\vdash,\ \Rightarrow\vdash,\ \vdash\Rightarrow,\ \text{axiom}$$

where the first rule to be tried is $\mathcal{D}\vdash$, and the last rule to be tried is axiom (one can think of it as an ordered list, although the underlying structure is capable of expressing much more complicated and specialized orders and strategies than this). This means that a term

is always expanded with its definition by the $\mathcal{D}\vdash$ rule, and another inference rule than $\mathcal{D}\vdash$ is used only when $\mathcal{D}\vdash$ cannot further reduce an expression.

Consider the goal sequent `add(s(0),s(0)) \- X`. The deduction tree is

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\texttt{X = 0}}}{\texttt{0 \- X}}}{\texttt{add(0,0) \- X}} \qquad \dfrac{\boxed{\texttt{Y = s(0)}}}{\texttt{s(X) \- Y}}}{\texttt{\- add(0,0) -> X}} \qquad \texttt{s(X) \- Y}}{\texttt{(add(0,0) -> X) -> s(X) \- Y}}}{\texttt{succ(add(0,0)) \- Y}}}{\texttt{add(s(0),0) \- Y}}$$

where `X` is instantiated to `0` when the axiom is applied to the sequent `0 \- X` of the left branch, and `Y` is instantiated to `s(0)` at the application of the axiom rule in the right branch.

3.3 Lazy evaluation

Lazy evaluation of an expression is accomplished by a search strategy that reorders the rules. Backtracking is used as the engine for computing more and more evaluated values.

To achieve lazy evaluation, the inference rules are reordered into the search strategy

axiom, $\Rightarrow\vdash, \vdash\Rightarrow, \mathcal{D}\vdash$

Consider again the goal `add(s(0),0) \- X`. The first answer is `X = add(s(0),0)`, since the axiom is tried first

$$\dfrac{\boxed{\texttt{X = add(s(0),0)}}}{\texttt{add(s(0),0) \- X}}$$

The next answer is `X = succ(add(0,0))`,

$$\dfrac{\dfrac{\boxed{\texttt{X = succ(add(0,0))}}}{\texttt{succ(add(0,0)) \- X}}}{\texttt{add(s(0),0) \- X}}$$

the next answer is `X = (add(0,0) -> Y) -> s(Y)`, and so forth until all steps in the tree in section 3.2 are computed.
It is actually the case that the reversed order of the answer substitutions of this section could be found by backtracking in section 3.2, with eager evaluation.

3.4 Making the definition complete

The definition in section 3.1 is not complete, since goals of the form

```
add(add(...,...),...) \- ...
```

are undefined. What is lacking is a clause to deal with unevaluated expressions in the first argument of `add`. To make the definition complete, we have to add a substitution schema to deal with something that is not `0` or `s(...)`. The complete definition is now

```
add(0,X) <= X.
add(s(Y),X) <= succ(add(Y,X)).
add(X,Y) <= [X ≠ 0, X ≠ s(_)], ((X -> Z) -> add(Z,Y)).

succ(X) <= ((X -> Y) -> s(Y)).
```

Note the guard in add's third clause. The guard is necessary to distinguish the third clause from the other two. The "_" symbol is a "don't care" variable.

We are now able to compute arbitrary expressions containing `add`-expressions and `succ`-expressions, for example

```
add(succ(add(s(0),0)),s(s(0))) \- X
```

which computes to `X = s(s(s(s(0))))`.

3.5 Equation solving

It is also the case that these definitions can be used for simple equation solving. The examples in this section make use of the search order defining eager evaluation.

Consider again the definition for add given in section 3.1. If we instead pose the query

```
add(s(0),X) \- s(s(0))
```

we get the binding `X = s(0)`, and no further answers. The equivalent query

```
add(X,s(0)) \- s(s(0))
```

also computes the binding `X = s(0)` first, but when we backtrack for another solution, the computation gets into a loop generating a deeper and deeper nested `s`-expression.

A more complex query is

```
(add(X,Y) \- s(s(s(0)))), (add(X,X) \- Y)
```

which computes `X` to `s(0)` and `Y` to `s(s(0))`.

The function difference can be defined in terms of add;

```
difference(X,Y) <= (add(Y,Z) -> X) -> Z.
```

This should be read as: if `add` of `X` and `Z` is `Y`, then the difference of `X` and `Y` is defined to be `Z`. As an example query, consider

```
difference(s(s(0)),s(0)) \- X
```

which computes `X` to `s(0)`. As this example shows, we are able to define a functions inverse in terms of the function itself. It is just a matter of instantiation.

It should be noted that with the formalism presented here, the execution will sometimes go into a loop when backtracking is used to find another answer. This could be avoided if the definition, or part of it, can be made deterministic. Typically, the query

```
add(X,0) \- s(0)
```

will, after the answer `X = s(0)`, go into an infinite loop where the system tries `X = s(s(0))`, `X = s(s(s(0)))`, etc. We hope that the control structure presented in [Kre91] will be able to deal with this kind of behaviour.

We have found this technique very useful, and among other things implemented a version of the planning system STRIPS [G&N], [Aro89], where an action is seen as a function from one state to another. It turns out that simulation and planning becomes just a matter of instantiation in the same manner as explained above; if the arguments to the action function are uninstantiated, but the conclusion is given, the system tries to find an action that results in the conclusion, i.e. planning, and if the actions are given but not the conclusion, the actions are evaluate to see the results, i.e simulation.

4. Mixing functional definitions and relational programming

As shown above, GCLA is able to execute both functional programs and logic programs beyond ordinary SLD resolution. It is also possible to combine functional and relational programming. The relational programs (i.e. horn clause programs) are executed by the rule $\vdash\mathcal{D}$ while functional programs are executed by the rules $\mathcal{D}\vdash$, $\vdash\Rightarrow$, $\Rightarrow\vdash$ and axiom. Note that the rule for executing relational programs together with the rules for executing functional program form the complete set of inference rules for GCLA (see section 2).

4.1. Program example 3: Qsort

The reason for having qsort as an example is that it is well known and can be implemented using a mix of relational and functional programming, not because we think that our version can be executed faster or something like that. It is just an example of an integration of functional and relational programming.
We assume that the relations =< and > on numbers are predefined.
The definition constitutes 9 clauses:

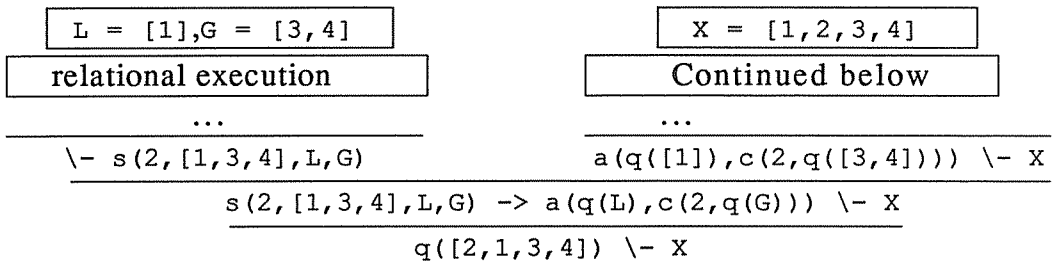
```
qsort([]) <= [].
qsort([F|R]) <=
      split(F,R,Less,Greater)
       -> append(qsort(Less),cons(F,qsort(Greater))).

split(X,[],[],[]).
split(X,[F|R],[F|L],G) <= F =< X,split(X,R,L,G).
split(X,[F|R],L,[F|G]) <= F > X,split(X,R,L,G).

append([],L) <= L.
append([F|R],L) <= cons(F,append(R,L)).
append(X,L) <= [X ≠ [], X ≠ [_|_]],
      ((X -> Y) -> append(Y,L)).

cons(X,Y) <= ((X -> Z),(Y -> W) -> [Z|W]).
```

`qsort` and `append` are functions, while `split` is a relation for computing the two halves `Less` and `Greater` in `qsort`.

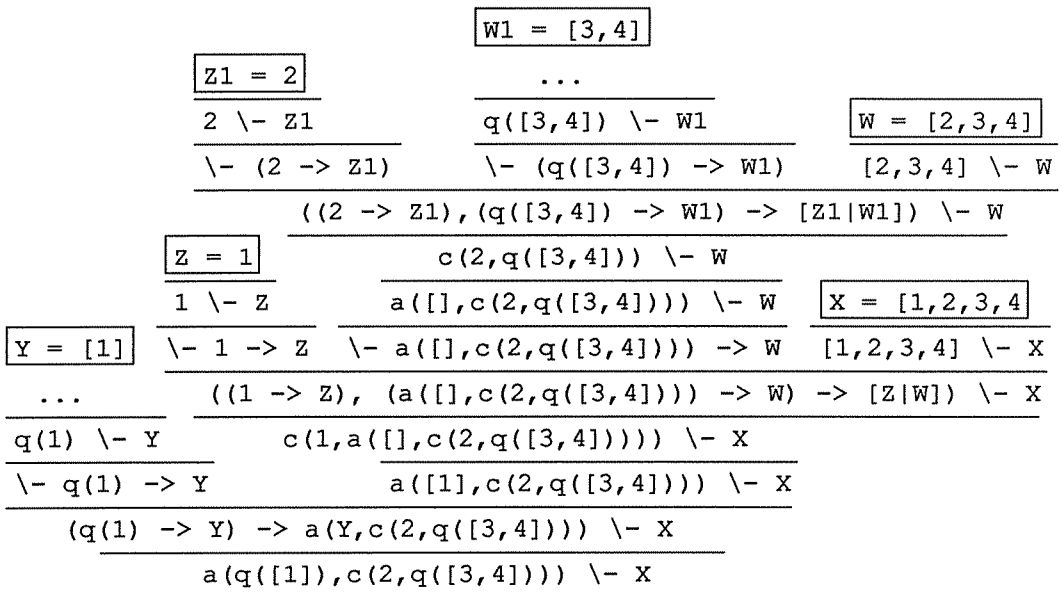
Consider the query

```
qsort([2,1,3,4]) \- X
```

which binds `X` to `[1,2,3,4]`. An interesting part of the deduction tree is where the "switching" between functional and relational execution takes place (we have abbreviated `append` with `a`, `qsort` with `q`, `cons` with `c` and `split` with `s`, and substituted variables with their values as soon as the value is known)



The left branch solves the goal `\- split(2,[1,3,4],L,G)` with applications of the rule $\vdash\mathcal{D}$, and then the right branch is solved, now with the two lists `L` and `G` instantiated.



4.2. Program example 4: Functional If-Then-Else

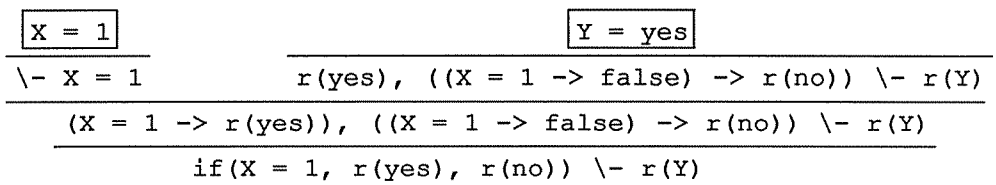
Another example is the common functional if-then-else construction, which also can be expressed as a mix of functional and relational programming. The definition is

```
if(If,Then,Else) <=
  (If -> Then), ((If -> false) -> Else).
```

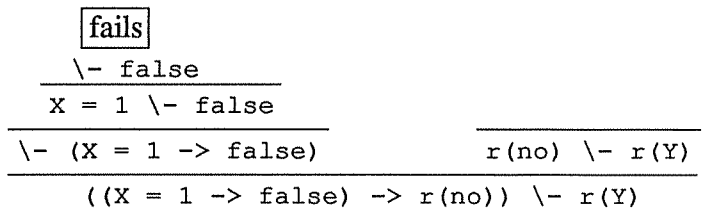
The condition `(X -> false)` is the way negation is accomplished in GCLA. `false` can be any symbol that is not defined in the program. One possible query is

```
if(X = 1,r(yes),r(no)) \- r(Y)
```

which instantiates `X` to `1` and `Y` to `yes`. The corresponding deduction tree is



To see that the assumption `((X = 1 -> false) -> r(no))` cannot reduce to `r(no)` is simple, consider the failed deduction



There is no possible combination of the inference rules making `\- (X = 1 -> false)` true.

It is also easy to see that if the `If` part is not satisfied, there is no possible way to reduce `(If -> Then)` to `Then`. For example, we cannot reduce `(2 = 1 -> r(yes))` to `r(yes)`. (It is in fact the case that `if` can fail all together, if neither `\- If` nor `If \- false` succeeds, which occur when the `If`-statement is neither provable by $\vdash_{\mathcal{D}}$ nor by $_{\mathcal{D}}\vdash$. This happens for example when the `If`-statement has been declared "circular" in the sense of section 2.3.)

It is also possible to backtrack into the `If`-part, if the `Then` part fails. This means that if the `If`-part can be solved in several ways, all these will be tested before the else-part is considered. This is in contrast to, for example most Prologs, where it is not possible to backtrack into the `If`-part.

Note that we have not used anything outside the theory; we have only used the ability to reason with hypothetical queries.

5. Conclusion

We have presented what a functional definition is, and how it can be used for both eager and lazy evaluation. It is possible to define other computation strategies as well, although we have not presented that here (see [Kre91] for a detailed description). We have also demonstrated how simple equation solving can be performed, and how a functions inverse can be defined in terms of the function itself.
GCLA is expressive enough to express both functional and relational programs within the same framework. An interesting step is to extend relational programming with the ability to assume things. The borders between relational and functional programming then disappears since both relational and functional programs uses the same set of inference rules, and we get a uniform framework for handling definitions instead of a framework, where the two branches (functional and relational) are incorporated into one another.
All examples given in this paper have been executed by our GCLA system, at the moment an interpreter written in Prolog. We have also implemented an earlier version of GCLA on top of an experimental abstract machine [Aro89], which was based on the WAM, and that implementation suggested that ordinary Prolog queries could be executed in GCLA in about 50 % of a traditional WAM implementation of Prolog.

6. References

[Aro89] Martin Aronsson; *GAM, An Abstract Machine for GCLA*, Research Report SICS R89002, Swedish Institute of Computer Science, 1989

[Aro90a] Martin Aronsson, Lars-Henrik Eriksson, Anette Gäredal, Lars Hallnäs, Peter Olin; *The programming Language GCLA: A Definitional Approach to Logic Programming*, New Generation Computing 7(4), pp 381 - 404, 1990

[Aro90b] Martin Aronsson, Lars-Henrik Eriksson, Lars Hallnäs, Per Kreuger; *A Survey of GCLA: A Definitional Approach to Logic Programming*, Extensions of Logic Programming: Proceedings of a workshop held at the SNS, Universität Tübingen, 8-9 December, 1989. Springer Lecture Notes in Artificial Intelligence

[DG-L86] D. DeGroot, G. Lindstrom; *Logic Programming, Functions Relations and Equations*, Prentice-Hall, 1986

[EH-88] Lars-Henrik Eriksson, Lars Hallnäs; *A Programming Calculus Based on Partial Inductive Definitions*, Research Report SICS R88013, Swedish Institute of Computer Science, 1988

[Fre90] Daniel Fredholm; *On Function Definitions. Basic notions and primitive recursive function definitions I*, Ph L thesis, Programming Methodology Group, Chalmers University of Technology, Göteborg

[HS-H88] Lars Hallnäs, Peter Schroeder-Heister; *A Proof-Theoretic Approach to Logic Programming*, Journal of Logic and Computation, 1990

[Hal87] Lars Hallnäs; *Partial inductive definitions*, in A. Avron et. al., editor, Workshop on General Logic, Report ECS-LFCS-88-52. Department of Computer Science, University of Edinburgh, 1987. Also published as Research Report SICS R86005C by the Swedish Institute of Computer Science, 1988. A revised version to appear in Theoretical Computer Science.

[Hen89] Pascal Van Hentenryck; *Constraint Satisfaction in Logic Programming*, MIT Press, MIT, Cambridge, USA

[Kre91] Per Kreuger, *GCLAII: A Definitional Approach to Control*, SICS Research Report in preparation