ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

A Functional Logic Language Based on Higher Order Narrowing

Herbert Kuchen

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Functional Logic Language Based on Higher Order Narrowing

Herbert Kuchen

RWTH Aachen, Lehrstuhl für Informatik II
D-52056 Aachen, Germany

Abstract

Functional logic languages have a syntax like a purely functional language but use narrowing as operational semantics. We present the functional logic language Higher Order Babel which provides higher order unification for parameter passing and solving equations. When searching for a function which solves an equation “polynomial functions” as well as defined functions are taken into account. In contrast to all other programming languages with higher order unification HO-Babel replaces the expensive β -reduction by the more efficient combinator reduction. Further, HO-Babel is more homogeneous since it does not distinguish functions which only represent data structures and defined function which have access to the full execution mechanism of the language.

1 Introduction

Integrated functional logic programming languages can be implemented more efficiently than purely logic languages, since the determinism of functions can be exploited to reduce the search space [LW91]. On the other hand, functional logic languages have more expressive power than purely functional languages, since they allow function inversion, logic variables and hence partial data structures (see [Ha94] for a survey on functional logic languages).

However, most existing functional logic languages only allow first order functions or at most to use higher order functions in a very restricted way, namely to pass arguments of a functional type by pattern matching rather than unification [KLMR90].

For meta-programming (i.e. program transformation and analysis) a kind of abstract data type λ -term with corresponding operations α -, β -, and η -conversion is helpful. If such a data type is available, the mechanisms of the programming language can be used to perform conversions on the object language level. A functional logic language with full support of higher order functions, including higher order unification, provides such a data type λ -term. Thus, well-known transformations like unfolding and the expansion of higher-order macros [Wa90] can be described in one line. See [HM88] for details and more examples.

On the other hand, higher order unification can be used to search for functions. This is especially interesting for theorem proving. If inference rules are represented by functions (e.g. from formulae to formulae), a proof amounts to searching for a suitable function (e.g. a mapping from axioms to the theorem). This function will be a composition of functions representing inference rules. We will use an example of this kind in order to present the language HO-Babel. More such examples can be found in [Fe92].

Other applications which benefit from higher order unification are type inference, natural language parsing, and formalizing logics (see [NM88, Pr94] for references).

Currently such applications are frequently implemented in λ -Prolog [NM88], an extension of Prolog by typed λ -terms and higher order unification. However, λ -Prolog mainly provides λ -terms as data structures, and it is not a fully integrated functional logic language. It is still difficult to detect deterministic computations and to handle them efficiently. Moreover, the search space for functions is limited to “polynomial functions”, i.e. functions which can be composed from constant functions and projections. Recursive functions, for instance, are not considered.

The advantages of functional languages and the expressivity of λ -Prolog can be combined by integrating higher order unification into a functional logic language. Moreover, this enables search in terms of the functions defined in the

program. A functional logic language with a limited unification of higher order variables is provided in [GHR92]. Extensions of functional logic languages by higher order unification have been proposed in [Pr94]. The latter approaches use λ -terms and $\beta\eta$ -normalization before unifying terms. λ -Prolog improves this a bit by implementing β -reduction lazily.

“Classical” β -reduction is known to be an expensive operation [Wa71]. If a function represented by a λ -term is applied to some argument, the body of the function has to be copied (due to possible sharing) and instantiated by the argument. This requires a scan over the (internal) representation of the body. Thus, in implementations of purely functional languages, the β -reduction has been replaced by combinator reduction [PJ87]. A combinator is a (named) closed λ -term. λ -terms can be translated to equivalent (variable free!) combinator terms, only consisting of combinator names (and possibly constants). A combinator application is reduced by directly applying the definition of the combinator as a rewrite rule. Since there are no variables, the copying and instantiation of the body of a function are avoided.

One reason that β -reduction is used in languages which provide higher order unification is that the equality of λ -terms, defined as the syntactic identity of their (long) $\beta\eta$ -normal forms (up to a renaming of variables; $\beta\eta$ -equality for short) is not preserved by the equality on the corresponding combinator terms, defined as the syntactic identity of the normal forms in the combinator calculus. This problem has been solved by Dougherty [Do93] and Hindley/Seldin[HS90], who have shown, how the $\beta\eta$ -equality on λ -terms can be simulated on SKI-combinator terms. Dougherty’s result can be generalized to other (sufficiently powerful) combinator systems.

This paper is organized as follows. In Section 2, the syntax of the functional logic language Higher Order Babel (HO-Babel) is presented, which is an extension of the “classical” Babel [MR92] by higher order unification based on a generalization of Dougherty’s approach. In Section 3, we define higher order narrowing as the operational semantics of HO-Babel. Section 4 contains the conclusions.

2 HO-Babel

HO-Babel is a functional logic language which has a syntax like a purely functional language but uses lazy (higher order) narrowing as operational semantics. In the present paper, we have adapted the syntax to (a subset of) Haskell [HPW92] since we assume most readers to be familiar with Haskell.

Instead of presenting the syntax formally, we give an example and explain the differences from Haskell. A formal description of the syntax of Babel rules, terms, and expressions can be found in the appendix.

The following program allows to prove theorems in group theory:

```
infixl 6 :*
data Term = One | Atom Integer | Term :* Term | Inv Term
infix 5 :=
data Equation = Term == Term

-- inference rules (subset required here)
lmult t (t' == t'') =      t :* t' == t :* t''
axiom t =                  t == t
invIn t (t' == t'') =      (Inv t) :* t :* t' == t''
invElim1 ((Inv t1) :* t2 :* t' == t'') | t1 == t2 =
                             t' == t''
invElim2 ((Inv t1) :* t2 == t') | t1 == t2 = One == t'
ass (t :* (t' :* t'') == t1) = t :* t' :* t'' == t1
swap (t == t') =           t' == t
```

A HO-Babel *program* consists of a sequence of data type (and fixity) definitions and a sequence of function definitions. A program can be queried with a goal expression. For example, a valid query for the above program is

```
invElim2.swap.invElim1.ass.(lmult (Inv (Inv (Atom 1)))).
(invIn (Atom 1)).axiom (Inv (Atom 1))
```

which delivers $\text{One} ::= (\text{Atom } 1) : * (\text{Inv } (\text{Atom } 1))$. From the point of view of group theory this means that by the above sequence of inference rules we have proved that $(\text{Inv } (\text{Atom } 1))$ is not only a left inverse of $(\text{Atom } 1)$ but also a right inverse ($:$ is the group operation with neutral element One). On the other hand, $(\text{Atom } 1)$ is (besides $(\text{Inv } (\text{Inv } (\text{Atom } 1)))$ a left inverse of $(\text{Inv } (\text{Atom } 1))$. A proof is represented by a composition of functions (inference rules) mapping equations (in group theory) to other equations. Since these functions reflect valid operations in group theory, the resulting equation holds provided that the argument equation holds.

The above goal can already be handled in Haskell. However, HO-Babel allows a goal like:

$$F (\text{Inv } (\text{Atom } 1)) == \text{One} ::= (\text{Atom } 1) : * (\text{Inv } (\text{Atom } 1))$$

where F is a *logic variable*. A *solution* of this goal consists of the *answer substitution*, in our case a binding of F to a combinator expression, which is equivalent to the composition of functions in the first goal, and the *result* `True`.

In contrast to Haskell, HO-Babel allows logic variables in goals and in the guard gd of a rule (i.e. a function binding in Haskell) $f t_1 \dots t_n \mid gd = e$. Here gd has to be a conjunction of equations, i.e. gd has to be of the form $e_1 == e'_1 \ \&\& \dots \ \&\& e_m == e'_m$ for $m \geq 0$. Equations of *functions* are valid, e.g. an equation like $F \ 0 == \text{mult } \text{One}$ is allowed. They will be solved by higher order unification.

If a rule $f t_1 \dots t_n \text{ rhs}$ can be applied this does not mean that an alternative rule $f p'_1 \dots p'_n \text{ rhs}'$ will not be considered. This may well occur after backtracking. Thus, in order to guarantee the determinism of functions we need stronger restrictions on function definitions than Haskell. Either the left hand sides of alternative rules are not unifiable or the corresponding right hand sides are syntactically identical after applying the most general unifier of the left hand sides. Moreover, alternative guarded right hand sides for a considered left hand side have to exclude each other. The compiler attempts to check this based on some simple syntactic tests.

For simplicity of presentation, partial applications of constructors are not allowed.

3 Operational Semantics

Next, we want to specify the operational semantics of HO-Babel. We choose a generalization of narrowing, where first order unification is replaced by a restricted form of higher order unification for parameter passing. Since this semantics shall provide the basis for the implementation of HO-Babel, it is more implementation oriented than operational semantics of similar languages. In particular, it is less non-deterministic and presents (the first order part of) narrowing in one step rather than splitting it into several rules. The semantics is only non-deterministic at points where the implementation will search via backtracking.

3.1 Preliminaries

Before we come to the actual definition of our operational semantics, we need some auxiliary notions. As mentioned in the introduction, we want to apply Dougherty's approach of simulating $\beta\eta$ -equality by an equality on "combinator expressions". For this approach, we need an additional set FC of (typed) *fill-in constants* which are handled similarly to constructors and which may occur in expressions (during computation).

An expression $e \in Expr$ is said to be in (weak) *head normal form* (HNF) (denoted: $hnf(e)$), if e has one of the following forms:

- $X e_1 \dots e_n$, with $X \in Var$
- $f e_1 \dots e_n$, with $f \in Fct, \text{arity}(f) > n$
- $c e_1 \dots e_n$, with $c \in Constr$; (note: $\text{arity}(c) = n$)
- $d e_1 \dots e_n$, with $d \in FC$

where Var is the set of logic variables, Fct the set of defined functions, $Constr$ the set of constructors, and $e_1, \dots, e_n$ are expressions.

An expression $e \in Expr$ in HNF is called *rigid* (denoted: $rigid(e)$), if $e := \varphi e_1 \dots e_n$ with $\varphi \in Constr \cup Fct \cup FC$ ($n \geq 0$), and *flexible* otherwise (denoted: $flexible(e)$). Correspondingly, an equation $e_1 == e_2$ is called a *flexible-flexible* (FF), *flexible-rigid* (FR), or *rigid-rigid* (RR) pair, depending on whether both, one or none of the expressions e_1 and e_2 is flexible.

Special FF- and FR-pairs of the form $X == e$ (or $e == X$), occurring e.g. when passing parameters, are handled like in the first order case, i.e. by binding X to e .

Other flexible-flexible pairs can also trivially be solved (e.g. by constant functions) [Hu75, SG89], but the corresponding set of unifiers is usually very large. Thus, following [Hu75, NJ89], we do not actually compute these unifiers but rather keep the FF-pairs as constraints during the computation.

Rigid-rigid pairs of the form $\varphi e_1 \dots e_n == \varphi e'_1 \dots e'_m$ with $\varphi \in Constr \cup FC$ are handled like in first order unification, i.e. they are decomposed.

FR-pairs of the form $F e_1 \dots e_n == \varphi e'_1 \dots e'_m$ ($\varphi \in Constr \cup FC$) where both sides of the equation have a non-functional type are most interesting from the point of view of higher order unification. The classical approach [Hu75] considers two possibilities for solving such an equation: *imitation* and *projection*.

Imitation attempts at binding F to

$$\lambda x_1, \dots, x_n. \varphi (H_1 x_1 \dots x_n) \dots (H_m x_1 \dots x_n)$$

where $H_1, \dots, H_m$ are new logical variables, i.e. it tries to solve the equation by providing the desired top-level constructor (here: $\varphi \in Constr$, see 3.3) of the rigid part and leaving the rest to the solution of subproblems.

Projection attempts at binding F to

$$\lambda x_1, \dots, x_n. x_i (H_1 x_1 \dots x_n) \dots (H_m x_1 \dots x_n)$$

for some i ($1 \leq i \leq n$) (m is chosen in such a way that the type of $\varphi e'_1 \dots e'_m$ is “compatible” with the type of $x_i (H_1 x_1 \dots x_n) \dots (H_m x_1 \dots x_n)$). Roughly, we try to generate the desired top-level symbol by an application of the i -th argument. The rest is again left to the solution of subproblems.

Let us consider an example: $F \ 1 == 1$. This equation is solved by¹ $\{F/\lambda x.1\}$, a special case of imitation, and by $\{F/\lambda x.x\}$, a particular projection.

In HO-Babel, we will also consider bindings of F to (partial applications of) defined functions. Another solution of the above example is e.g. based on the factorial function (i.e. $\{F/fac\}$, where $fac \ x = \text{if } x == 0 \text{ then } 1 \text{ else } x * fac \ (x-1)$).

If we assume that appropriate functions for simulating imitation and projections are defined (by the user or provided by the compiler), the guessing of defined functions is sufficient and we do not have to consider imitations and projections explicitly. Note that defined functions may be recursive and/or partial, while imitation and projection only allow to consider “polynomial functions”.

Other FR- and RR-pairs of the form $e_1 = e_2$, where e_1 (and hence also e_2) has a functional type, are handled using Dougherty’s approach [Do93]. The main idea is to transform $e_1 == e_2$ to $e_1 \ d == e_2 \ d$ for some “new” *fill-in constant* $d \in FC$. Since no assumptions are made about the actual value of d , this essentially implies that e_1 and e_2 are equal for all arguments d and thus equal to each other (extensionality). The details of applying this idea will be discussed later.

Let us first come back to the modified first order unification mentioned above. As in [NJ89], the idea is to handle the “first order part” of a unification first and to delay the treatment of encountered FR-pairs, since the first order part can be handled quite efficiently while the treatment of FR-pairs requires search. The searching effort is wasted, if the first order part of the unification fails anyway. Formally, the unification is described by the schemes *unify* and *unify'* shown below.

The scheme *unify* handles the first order part of the unification for parameter passing. For a term t (on the left hand side of a rule) and an expression e (the corresponding actual argument) sharing no variables with t , *unify* (t, e)

¹A substitution is denoted by a set of pairs $\{X_1/e_1, \dots, X_k/e_k\}$. Each pair X_i/e_i tells that variable X_i is substituted by e_i . $\emptyset$ is the identical substitution. $x \ (\sigma \circ \mu) := (x\sigma)\mu$.

computes a triple (μ, σ, E) consisting of a substitution μ (for t), a substitution σ (for e), and a set of remaining equations of type flexible-rigid. If e is a first order expression, $\sigma \circ \mu$ is the most general unifier of t and e .

$$\begin{aligned}
\text{unify}(c\ t_1 \dots t_n, c\ e_1 \dots e_n) &:= \\
&\text{let } \sigma'_0 := \emptyset, \\
&\quad (\mu_i, \sigma_i, E_i) := \text{unify}(t_i, e_i \sigma'_{i-1}) \\
&\quad \sigma'_i := \sigma'_{i-1} \circ \sigma_i \quad i = 1, \dots, n \\
&\text{in } (\bigcup_{i=1}^n \mu_i \circ \sigma'_n, \sigma'_n, \bigcup_{i=1}^n E_i \sigma'_n) \\
\text{unify}(X, e) &:= (\{X/e\}, \emptyset, \emptyset) \\
\text{unify}(t, X) &:= (\emptyset, \{X/t\}, \emptyset) \quad \text{for } t \notin \text{Var} \\
\text{unify}(c\ t_1 \dots t_n, F\ e_1 \dots e_m) &:= (\emptyset, \emptyset, \{c\ t_1 \dots t_n == F\ e_1 \dots e_m\}) \quad \text{for } F \in \text{Var}, m > 0
\end{aligned}$$

Note that a variable of functional type on the left hand side is just substituted by the corresponding expression, while an application of a variable in the goal leads to a new equation (of type flexible-rigid) whose solution is postponed until the first order part of the unification is finished. Note also that the unification fails (*unify* is undefined), if e is not sufficiently evaluated. For instance, $\text{unify}(c\ t_1 \dots t_n, f\ e_1 \dots e_m)$ is undefined, since a comparison with the term $c\ t_1 \dots t_n$ would require $(f\ e_1 \dots e_m)$ to be evaluated. $(f\ e_1 \dots e_m)$ cannot be a partial application, since our type system ensures that only terms and expressions of appropriate types are unified and terms cannot be of a functional type (partial applications of constructors are not allowed).

The scheme *unify'* handles the first order part of a general higher order unification required for the solution of an equation in a guard of a rule. For two expressions e and e' in HNF, *unify'* (e, e') computes a substitution σ (which has to be applied to e and e') and a set of remaining equations of type flexible-flexible and flexible-rigid. If e and e' are first order expressions, σ is the most general unifier.

$$\begin{aligned}
\text{unify}'(\varphi\ e_1 \dots e_n, \varphi\ e'_1 \dots e'_n) &:= \quad \text{for } \varphi \in \text{Constr} \cup FC \\
&\text{let } \sigma'_0 := \emptyset, \\
&\quad (\sigma_i, E_i) := \text{unify}'(e_i \sigma'_{i-1}, e'_i \sigma'_{i-1}) \\
&\quad \sigma'_i := \sigma'_{i-1} \circ \sigma_i \quad i = 1, \dots, n \\
&\text{in } (\sigma'_n, \bigcup_{i=1}^n E_i \sigma'_n) \\
\text{unify}'(X, e) &:= (\{X/e\}, \emptyset) \quad \text{if } e \text{ is } X \text{ (syntactically)} \\
\text{unify}'(e, X) &:= (\{X/e\}, \emptyset) \quad \text{if } e \notin \text{Var} \\
\text{unify}'(e, e') &:= (\emptyset, \{e == e'\}) \quad \text{if no previous case applies,} \\
&\quad \text{(note: } e == e' \text{ is flexible-flexible or flexible-rigid then)}
\end{aligned}$$

3.2 Higher Order Narrowing

With these auxiliary functions, we can now formalize the operational semantics of HO-Babel. We define a family of single step relations $\Rightarrow_\sigma$ acting over pairs consisting of the (current) goal and the current set of equations, which serve as a kind of constraints for the solution. Here σ is the substitution which has to be applied when transforming one pair into another.

(NA) Narrowing:

$$\begin{aligned}
&\langle f\ e_1 \dots e_n, E \rangle \Rightarrow_\sigma \langle e\mu, E'' \cup E'\mu \cup E\sigma \rangle \\
&\text{if there is a (variant of a) program rule} \\
&\quad f\ t_1 \dots t_n \mid E' = e \text{ with “fresh” variables, and} \\
&\quad (\mu, \sigma, E'') := \text{unify}(c_f\ t_1 \dots t_n, c_f\ e_1 \dots e_n), \text{ and} \\
&\quad E \text{ contains only FF-pairs.}
\end{aligned}$$

c_f is an auxiliary constructor of the same type as f which is introduced here for technical reasons. Note that due to the definition of *unify* this rule cannot be applied, if the expressions $e_1, \dots, e_n$ are insufficiently evaluated.

(AA) Add argument:

$$\begin{aligned} \langle e, \{e_1 == e_2\} \cup E \rangle &\Rightarrow_{\emptyset} \langle e, \{e_1 d == e_2 d\} \cup E \rangle \\ \text{if } type(e_1) \text{ is functional, } hnf(e_1), hnf(e_2), rigid(e_1) \text{ or } rigid(e_2) \\ d \text{ is a "new" fill-in constant with } type(d) = \tau_1, \text{ if } type(e_1) = \tau_1 \rightarrow \tau_2. \end{aligned}$$

(SE) Solving equations:

$$\begin{aligned} \langle e, \{e_1 == e_2\} \cup E \rangle &\Rightarrow_{\sigma} \langle e\sigma, E' \cup E\sigma \rangle \\ \text{if } type(e_i) \text{ is not functional, } hnf(e_i), rigid(e_i) \text{ or } e_i \in Var \text{ for } i \in \{1, 2\} \\ (\sigma, E') &:= unify'(e_1, e_2) \end{aligned}$$

Note that this rule cannot be applied, if *unify'* fails due to unsufficiently evaluated expressions.

(ES) Evaluation of subexpression:

$$\begin{aligned} \langle \varphi e_1 \dots e_n, E \rangle &\Rightarrow_{\sigma} \langle \varphi (e_1\sigma) \dots e'_i \dots (e_n\sigma), E' \rangle \\ \text{if } \varphi \in Constr \cup Fct \cup FC, \langle e_i, E \rangle &\Rightarrow_{\sigma} \langle e'_i, E' \rangle, \text{ for } i \in \{1, \dots, n\} \end{aligned}$$

This rule may only be applied, if the evaluation of e_i is “demanded” by some outer computation, e.g. in order to make rule (NA) applicable.

(EE) Evaluation of expression in equation:

$$\begin{aligned} \langle e, \{e_1 == e_2\} \cup E \rangle &\Rightarrow_{\sigma} \langle e\sigma, \{e'_i == e_{3-i}\sigma\} \cup E' \rangle \\ \text{if } \langle e_i, E \rangle &\Rightarrow_{\sigma} \langle e'_i, E' \rangle, \text{ for some } i \in \{1, 2\} \end{aligned}$$

This rule may only be applied, if this is necessary in order to make rule (SE) or rule (AA) applicable.

The following rule for guessing defined functions is needed for the actual higher order unification. Note that this covers imitation and projection, if appropriate functions for their simulation have been defined. Note also that FF-pairs are not transformed.

(GF) Guess defined function:

$$\begin{aligned} \langle e, \{F e_1 \dots e_n == \varphi e'_1 \dots e'_m\} \cup E \rangle &\Rightarrow_{\sigma} \langle e, \{F e_1 \dots e_n == \varphi e'_1 \dots e'_m\} \cup E \rangle \sigma \\ \text{if } \sigma &:= \{F/guess_{f,l,n} H_1 \dots H_l\}, H_i \text{ "new" for } i = 1, \dots, l, \varphi \in FC \cup Constr \\ \text{where } guess_{f,l,n} &\text{ is defined by the rule:} \\ guess_{f,l,n} X_1 \dots X_l Y_1 \dots Y_n &= f(X_1 Y_1 \dots Y_n) \dots (X_l Y_1 \dots Y_n) \\ \text{and } f \text{ is a } k\text{-ary defined function } (k \geq l \geq 0) &\text{ of appropriate type.} \end{aligned}$$

The guess rule tries to solve a FR-pair by guessing a defined function. We implicitly assume a variant of the (GF) rule, where the left and right hand sides of the considered equations are interchanged. Note that the auxiliary functions $guess_{f,l,n}$ can be statically determined and supplied by the compiler. In particular, n can be limited by the maximal arity occurring in the program. Note that the number of arguments of the newly introduced variables $H_1, \dots, H_l$ is equal to the number of arguments of the variable F .

In the same situation, the (GF) rule may be applied in several ways. Thus, in the implementation we have to select one way how to apply it nondeterministically. The other alternatives will be tried after backtracking.

A *computation* is a sequence of steps of the form:

$$\langle goal, \emptyset \rangle \Rightarrow_{\sigma_1} \dots \Rightarrow_{\sigma_n} \langle t, E \rangle$$

where $t \in Term$ and E only contains FF-pairs. t is called the *result* of the computation, $\sigma := \sigma_1 \circ \dots \circ \sigma_n$ is the *answer (substitution)* and E is the set of remaining constraints.

In order to demonstrate how these rules work together, we will show a few steps of the evaluation for the example equation from Section 2:

$\langle \text{True}, \{F(\text{Inv}(\text{Atom } 1)) == \text{One} := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1))\} \rangle \Rightarrow_{\{F/\text{guess}_{\text{invElim2},1,1}\}} (\text{GF}, \text{EE}, \text{NA})$
 $\langle \text{True}, \{\text{invElim2}(\text{H1}(\text{Inv}(\text{Atom } 1))) == \text{One} := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1))\} \rangle \Rightarrow_{\emptyset} (\text{EE}, \text{NA})$
 $\langle \text{True}, \{\text{One} := t' == \text{One} := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1)), t1 == t2, \\ \text{H1}(\text{Inv}(\text{Atom } 1)) == (\text{Inv } t1) : * t2 := t'\} \rangle \Rightarrow_{\{t'/(\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1)), t1/t2\}} (\text{SE}, \text{SE})$
 $\langle \text{True}, \{\text{H1}(\text{Inv}(\text{Atom } 1)) == (\text{Inv } t2) : * t2 := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1))\} \rangle \Rightarrow_{\{H1/\text{guess}_{\text{swap},1,1}\}} (\text{GF}, \text{EE}, \text{NA})$
 $\langle \text{True}, \{\text{swap}(\text{H2}(\text{Inv}(\text{Atom } 1))) == (\text{Inv } t2) : * t2 := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1))\} \rangle \Rightarrow_{\emptyset} (\text{EE}, \text{NA})$
 $\langle \text{True}, \{t3 := t4 == (\text{Inv } t2) : * t2 := (\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1)), \\ \text{H2}(\text{Inv}(\text{Atom } 1)) == t4 := t3\} \rangle \Rightarrow_{\{t3/(\text{Inv } t2) : * t2, t4/(\text{Atom } 1) : * (\text{Inv}(\text{Atom } 1))\}} (\text{SE})$
 $\vdots$
 After several further steps we get:
 $\langle \text{True}, \emptyset \rangle$

i.e. result True with the answer (restricted to the variables from the goal) $F/(\text{guess}_{12,1,1}(\text{guess}_{\text{swap},1,1}(\text{guess}_{11,1,1}(\text{guess}_{\text{ass},1,1}(\text{guess}_{\text{mult},2,2}(\text{guess}_{\text{inv},1,1}(\text{guess}_{\text{proj},1,1}(\text{guess}_{\text{invIn},2,2}(\text{guess}_{\text{proj},1,1}(\text{guess}_{\text{axiom},1,1}))))))))$ which essentially is a combinator representation of the function used in the first goal in Section 2. Here, *inv* and *proj* are auxiliary functions needed to simulate an imitation and projection, respectively:

$$\text{inv } x = \text{Inv } x \quad \text{and} \quad \text{proj } x = x$$

3.3 Fill-in Constants

If the above rules are applied naively, fill-in constants may appear in the result and/or answer substitution. This violates the soundness of the computation, since it means that (implicitly universally quantified) fill-in constants escape from their scope and appear in a context, where they have no meaning. Consider the equation $f \ X == \text{const } 0$ (which has no solutions) and the following rules:

$$f \ x \ y \mid x == y = 0 \\ \text{const } x \ y = x$$

Using explicit quantifiers, the above equation corresponds to

$$\exists X \forall d \ f \ X \ d == \text{const } 0 \ d.$$

Let us consider the following state including the equation and a corresponding computation:

$$\begin{array}{ll}
 \langle X, \{f \ X == \text{const } 0\} \rangle \Rightarrow_{\emptyset} & (\text{AA}) \\
 \langle X, \{f \ X \ d == \text{const } 0 \ d\} \rangle \Rightarrow_{\emptyset} \circ \Rightarrow_{\emptyset} & (\text{EE}, \text{NA}, \text{EE}, \text{NA}) \\
 \langle X, \{0 == 0, X == d\} \rangle \Rightarrow_{\emptyset} \circ \Rightarrow_{\{X/d\}} & (\text{SE}, \text{SE}) \\
 \langle d, \emptyset \rangle &
 \end{array}$$

Thus, the result is *d* and the answer substitution $\{X/d\}$. *d* has escaped from its scope. This has, of course, to be avoided. Technically, this problem can be solved by assigning a *quantifier level* to each variable and to each fill-in constant. Then, a variable *X* may only be substituted by an expression *e*, if *e* does not contain a fill-in constant *d* with a higher (i.e. inner) quantifier level than *X* (see [KA96] for implementation details).

4 Conclusions and Future Work

We have presented the functional logic language HO-Babel which supports higher order unification. In contrast to λ -Prolog, it allows to search in terms of the defined functions and provides hence a more powerful search mechanism. In contrast to all other programming languages with higher order unification, it eliminates the distinction of functions which are used as data structures only (typically represented by λ -terms) and functions/predicates which are equipped with the full execution mechanism of the language. Thus, HO-Babel is more homogeneous than these languages.

Equally important is that the replacement of the expensive β -reduction by combinator reduction allows, as in purely functional languages, a more efficient implementation.

We have presented an operational semantics for HO-Babel based on a generalization of lazy narrowing. This operational semantics is the basis for the implementation of HO-Babel.

For certain applications the search space for higher order unification may be quite large. One possibility to reduce it is to allow annotations by the user which restrict the applicability of the inference rules. In particular the user could exclude several defined functions from consideration when trying to apply the (GF) rule. We need more experience with practical programs in order to get a feeling for how this could be done in a reasonable way.

Moreover, our search mechanism will deliver several ($\beta\eta$) equivalent solutions, e.g. $\{F/(S\ K\ I)\}$ and $\{F/I\}$ both solve $F = I$ (where S , K , and I are the well-known combinators (see e.g. [PJ87])). We want to study, how far it is possible to reduce this kind of redundancy.

References

- [Do93] D.J. Dougherty: Higher-Order Unification via Combinators, TCS 114, 273-298, 1993.
- [Fe92] A. Felty: A Logic Programming Approach to Implementing Higher-Order Term Rewriting, 2nd Int. Workshop Extended Logic Programming, LNAI 596, Springer, 135-161, 1992.
- [GHR92] J.C. González-Moreno, M.T. Hortalá-González, M. Rodríguez-Artalejo: On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming, Proceedings CSL'92, LNCS 702, Springer, 216-230, 1993.
- [Ha94] M. Hanus: The Integration of Functions into Logic, From Theory to Practice, Journal of Logic Programming 19&20, 583-628, 1994.
- [HM88] J. Hannan, D. Miller: Uses of Higher-Order Unification for Implementing Program Transformers, Proceedings 5th ICLP, 942-959, MIT Press, 1988.
- [HPW92] P. Hudak, S. Peyton Jones, P. Wadler (eds.): Report on the Programming Language Haskell: A Non-Strict Purely Functional Language, SIGPLAN Notices 27(5), 1992.
- [Hu75] G.P. Huet: A Unification Algorithm for Typed λ -Calculus, TCS 1, 27-57, 1975.
- [HS90] J.R. Hindley, J.P. Seldin: Introduction to Combinators and λ -Calculus, Cambridge University Press, 1990.
- [KA96] H. Kuchen, J. Anastasiadis: Higher Order Babel — Language and Implementation, Proceedings ELP'96, LNCS, Springer Verlag, to appear.
- [KLMR90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-based Implementation of a Functional Logic Language, Procs. ESOP'90, LNCS 432, 1990, 271-290.
- [LW91] R. Loogen, S. Winkler: Dynamic Detection of Determinism in Functional Logic Languages, Procs. PLILP'91, LNCS 528, 1991, 335-346.
- [MR92] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language Babel, J. Logic Programming, 12, 189-223 (1992).
- [NJ89] G. Nadathur, B. Jayaraman: Towards a WAM Model for λ -Prolog, Proceedings of ICLP, MIT Press, 1180-1198, 1989.
- [NM88] G. Nadathur, D. Miller: An Overview of λ -Prolog, Proceedings 5th ICLP, MIT Press, 810-827, 1988.
- [PJ87] S. Peyton Jones: The Implementation of Functional Programming Languages, Prentice-Hall, 1987.
- [Pr94] C. Prehofer: Higher Order Narrowing, Proceedings LICS'94, IEEE, 507-516, 1994.

[SG89] W. Snyder, J. Gallier: Higher-Order Unification Revisited: Complete Sets of Transformations, J. Symbolic Computation 8, 101-140, 1989.

[Wa71] C.P. Wadsworth: Semantics and Pragmatics of the λ -Calculus, Thesis, Oxford'71.

[Wa90] P. Wadler: Deforestation: Transforming Programs to Eliminate Trees, TCS 73, 231-248, 1990.

A Syntax of HO-Babel Rules

For each function (symbol), its type and a sequence of defining rules are specified. A *rule* for a function f has the form

$$\underbrace{f \underbrace{t_1 \dots t_m}_{\text{left hand side}}}_{\text{left hand side}} \quad \underbrace{\left| \underbrace{E}_{\text{optional guard}} \right.}_{\text{optional guard}} \quad = \quad \underbrace{e}_{\text{body}} \quad \underbrace{\hspace{10em}}_{\text{right hand side}}$$

where $t_1, \dots, t_n$ are terms (also called: patterns), E is a sequence of equations $e_1 == e'_1 \&\& \dots \&\& e_m == e'_m$, and $e, e_1, e'_1, \dots, e_m, e'_m$ are arbitrary expressions. We require all rules for the same function f to have the same number of terms on the left hand side. This number is called *arity*(f).

A *term* ($t \in \text{Term}$) is either a *variable* (taken from a countable set Var of variables; denoted by an identifier beginning with a capital letter) or an application of a n -ary data constructor ($n \geq 0$) to n argument terms:

$$\begin{array}{ll} t ::= X & \% \text{ variable} \\ \quad | (c \ t_1 \dots t_n) & \% \text{ application of a } n\text{-ary data constructor } c \ (n \geq 0) \end{array}$$

An *expression* ($e \in \text{Expr}$) has the form:

$$\begin{array}{ll} e ::= X & \% \text{ variable} \\ \quad | (c \ e_1 \dots e_n) & \% \text{ application of a } n\text{-ary data constructor } c \ (n \geq 0) \\ \quad | f & \% \text{ function } f \\ \quad | (e \ e') & \% \text{ application of } e \text{ to } e' \end{array}$$

All terms and expressions have to be *well typed* (like in Haskell).