Implementing Disequality in the Lazy Functional Logic Language Babel

Herbert Kuchen
RWTH Aachen*

Francisco Javier López-Fraguas
Universidad Complutense de Madrid†

Juan José Moreno-Navarro
Universidad Politécnica de Madrid‡

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid§

Abstract

In this paper[1], we investigate an implementation of a lazy functional logic language (in particular the language BABEL [MR88,MR92]) which uses disequality constraints for solving equations and building answers. We specify a new operational semantics which combines lazy narrowing with disequality constraints and we define an abstract machine tailored to the execution of BABEL programs according to this semantics. The machine is designed as a quite natural extension of a lazy graph narrowing machine [MKLR90]. Disjunctions of disequalities are handled using the backtracking mechanism.

1 Introduction

During the last years, many proposals for combining the functional and logic programming paradigms have been made [BL86,DL86]. In particular, so called *functional logic languages* [Re85] retain functional syntax but use *narrowing* – a unification based parameter passing mechanism which subsumes rewriting and SLD resolution – as operational semantics.

There are also several works aiming at the efficient implementation of logic + functional languages by means of abstract machines supporting combinations of functional and logic programming capabilities [BBCMMS89,BCGMP89,Ha90,Mu90,KLMR90,MKLR90,Lo91]. In [KLMR90,MKLR90], we have investigated implementations of the lazy functional logic language BABEL [MR88,MR92] on the basis of *graph narrowing machines*, i.e. graph reduction machines extended by additional mechanisms for supporting logic variables, unification and backtracking.

Here, we study techniques for handling disequality constraints in BABEL. In particular, disequations of the form $X \neq t$ will be used when expressing answers.

It is well known that answer substitutions can be regarded as sets of equality constraints $X = t$. Allowing for disequalities increases the *expressivity* substantially. For instance, the disequation $X \neq Y$ cannot be replaced by any equivalent, finite set of equations. Disequations of this kind are not well supported by non-constraint-based languages, as in the case of previous versions of BABEL (where a disequality is in general a source of infinitely many computations all giving substitutions as answers) or in PROLOG (where $\neq$ is a built-in predicate which simply checks for non-unifiability, but can neither produce variable bindings nor occur in answers).

There is a deep theoretical work on solving *equational problems*, which include both equations and disequations over a Herbrand universe as particular cases [Co90,CL89]. On the other hand, constraint solving has been integrated into many PROLOG-like logic programming languages [Co82,Co84,HS88,JL87,DVSAGB88]. In particular, the use of disequality constraints is a quite common feature in these languages as a useful programming tool. We are currently working out a general scheme for *Constraint Functional Logic Programming* [LR91,Lo92] which should help to understand the work in this paper from a broader perspective. This theoretical framework will be suitable for obtaining soundness and completeness results for the language presented here.

The rest of the paper is organized as follows. In Section 2 we define *uniform BABEL programs*, which were introduced in [MKLR90]. Section 3 discusses the motivations that led us to the representation of

*Lehrstuhl für Informatik II, Ahornstr. 55, D-5100 Aachen, Germany, email: herbert@zeus.informatik.rwth-aachen.de

†Departamento de Informática y Automática, Avenida Complutense s/n, 28040 Madrid, Spain, email: fraguas@dia.ucm.es

‡Departamento LSIIS, Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain, email: jjmoreno@fi.upm.es

§Departamento de Informática y Automática (see above), email: mario@emducm11.bitnet

[1] The present paper is a detailed version of [KLMR92b].

constraints used in our implementation. In Section 4, we give a formal specification of BABEL's operational semantics. In particular, we develop a generalization of lazy narrowing taking into account disequality constraints. Section 5 presents the design of an abstract machine which extends the lazy graph-narrowing from [MKLR90] and implements the new operational semantics. This implementation uses the existing backtracking mechanism to handle backtracking due to constraint solving. In Section 6, we summarize and point out future work.

2 Uniform BABEL Programs

In this section, we present the functional logic language which is used in the rest of the paper and we specify its syntax. It is the *uniform, first order* fragment of the language BABEL [KLMR90]. In this paper, we restrict ourselves to first order programs for simplicity. An extension to higher order programs could follow the ideas outlined in [MKLR90]. Constraints would affect only first order variables.

Uniformity [MKLR90] is a syntactic restriction, which allows a more efficient implementation of lazy narrowing. More precisely, backtracking due to different redexes can be replaced by the usual backtracking due to different rules.

2.1 Overview

Uniform First Order BABEL programs (UFO-BABEL programs, for short) consist of declarations for *data types* (together with the corresponding *data constructors*) and definitions for *functions*. *Predicates* are viewed as *boolean functions*. *Horn Clause Logic Programs* can be translated into BABEL programs as explained in [MR92,KLMR90].

In this paper, we deal with a *computation mechanism* which can be roughly described as follows: A *goal* for a program may be any expression. Goals are intended to be reduced to a *result* – a data term – by computations which also compute some *answer constraints* – both *equational* and *disequational* – for the initial variables. This differs from the operational semantics in our previous works [KLMR90,MKLR90], where BABEL programs could compute only equality constraints (represented by substitutions) as answers. The following program computes the *size* of a list, understood as the number of different elements:

fun *member*: $\alpha \times (list\ \alpha) \rightarrow$ bool.
  member Y nil := *false*.
  member Y (cons X Xs) :=
    $X = Y \rightarrow$ *true* $\square$
        member Y Xs.

fun *size*: $(list\ \alpha) \rightarrow nat$.
  size nil := 0.
  size (cons X Xs) :=
    member X Xs $\rightarrow$ *size* Xs $\square$
        suc (*size* Xs).

solve *size* (*cons* (*mkpair* X 0) (*cons* (*mkpair* Y Z) nil)).
> 1. result: *suc* 0   answer: $X = Y$, $Z = 0$;
> 2. result: *suc* (*suc* 0)   answer: $X \neq Y$;
> 3. result: *suc* (*suc* 0)   answer: $Z \neq 0$;
> no more solutions

Note that the conditional expression in the second rule for *member* implicitly introduces the disequality constraint $X \neq Y$, if the else-branch is selected. Also note that the answer constraint "$X \neq Y$" cannot be replaced by equality constraints (i.e. by a single substitution). Previous versions of BABEL would compute infinitely many solutions for this goal. As far as we know, the same holds for other existing logic + functional languages.

In the following, we explain the syntax of UFO-BABEL programs more formally.

2.2 Data Types and Data Constructors

We assume a ranked set $TC = \cup_{n \in I\!N} TC^n$ of *type constructors* δ/n (e.g. nat/0, list/1) and a countably infinite set *TVar* of *type variables* α, β etc. Any algebraic term τ built from type constructors and type variables – e.g. (list nat), (list α) – is a *data type*. A data type is *polymorphic* if it includes type variables, and *monomorphic* otherwise. We also assume a set DC of *data constructors* with declared principal types: $c : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, e.g. *cons*: $\alpha \times (list\ \ \alpha) \rightarrow list\ \alpha$. In practice, type constructors and data constructors can be introduced through **datatype** declarations; e.g. the following data types, which are predefined in

BABEL:

datatype *bool*	$:=$ *true* $\mid$ *false*.	**datatype** *list* α	$:=$ *nil* $\mid$ *cons* α (*list* α).
datatype *nat*	$:= 0 \mid$ *suc nat*.	**datatype** *pair* $\alpha\ \beta$	$:=$ *mkpair* $\alpha\ \beta$.

2.3 Terms and Expressions

Next, we assume a countably infinite set *Var* of (data) variables and a set *FS* of *function symbols* with declared principal types: $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, for example *size* : *list* α $\rightarrow$ *nat*. We can then build well typed *terms* t and *expressions* e. We understand *well typedness* in the sense of Milner's type system [Mi78].

$$\begin{array}{lll l l} t & ::= & X & \% & X \in Var \\ & \mid & (c\ t_1 \ldots t_n) & \% & c \in DC \end{array} \qquad \begin{array}{lll l} e & ::= & X & \%\ X \in Var \\ & \mid & (c\ e_1 \ldots e_n) & \%\ c \in DC. \\ & \mid & (f\ e_1 \ldots e_n) & \%\ f \in FS. \end{array}$$

where $c : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, and $n \geq 0$. We assume that application associates to the left and omit parentheses accordingly. The syntax allows to build expressions involving some *primitive function symbols*, assumed to be present in *FS* and used as prefix, infix and mixfix operators (we assume b, b_i : *bool*): $\neg b$ (negation), $(b_1 \wedge b_2)$ (conjunction), $(b_1 \vee b_2)$ (disjunction), $(b \rightarrow e)$ (guarded expression, meaning: **if** b **then** e **else** undefined), $(b \rightarrow e_1 \Box e_2)$ (conditional, meaning: **if** b **then** e_1 **else** e_2), and $(e_1 = e_2)$ (weak equality). A *weak equation* $(e_1 = e_2)$ is intended to be *true*, if e_1 and e_2 have the same finite and totally defined value (infinite and/or partially defined objects are possible, since our intended semantics for data constructors is not strict). $(e_1 = e_2)$ is intended to be *false*, if the values for e_1 and e_2 differ in some constructor (even if e_1, e_2 represent infinite or partially defined objects). For the *declarative semantics* of weak equality, we refer the reader to [MR88,MR92]. An *operational semantics*, based on equality and disequality constraints, will be formally specified in Section 4.

2.4 Uniform Programs

UFO-BABEL programs consist of declarations of datatypes and *shallow defining rules* for function symbols. For a function symbol $f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, each defining rule must have the following shape:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left hand side}} \;:=\; \underbrace{\underbrace{\{b \rightarrow\}}_{\text{optional guard}}\ \underbrace{e}_{\text{body}}}_{\text{right hand side}}$$

The following restrictions must be satisfied:

1. *Shallow Data Patterns*: Each t_i is either a variable X or a shallow pattern $(c\ X_1 \ldots X_n)$ with $X_i \in Var$ for $i = 1, \ldots, n$ $(n \geq 0)$. In the latter case, we say that f *demands* its i-th argument, and we call c the *demanded constructor*.
2. *Left Linearity*: $f\ t_1 \ldots t_n$ does not contain multiple variable occurrences.
3. *Well Typedness*: Under appropriate type assumptions for the variables, it must be possible to check the types τ_i for each t_i $(1 \leq i \leq n)$, the type *bool* for b, and the type τ for e.
4. *Restrictions on Free Variables*: Any variable that occurs only in the right hand side is called *free*. Free variables are allowed to occur in the guard, but not in the body.

Moreover, the set of defining rules for each function symbol f must satisfy two additional requirements. Given two different defining rules for the same symbol f:

$$f\ t_1 \ldots t_n := \{b_1 \rightarrow\} e_1 \qquad\qquad f\ s_1 \ldots s_n := \{b_2 \rightarrow\} e_2$$

(with variables renamed apart), we require

5. *Uniformity*: for $1 \leq i \leq n$, t_i is a variable iff s_i is a variable.
6. *Nonambiguity*: if $(f\ t_1 \ldots t_n)$ and $(f\ s_1 \ldots s_n)$ are unifiable with most general unifier σ, then either $e_1\sigma$ and $e_2\sigma$ are identical, or $b_1\sigma$ and $b_2\sigma$ are *incompatible*. For a definition of incompatibility, we refer to [KLMR92]. The conjunction of two incompatible boolean expressions is always unsatisfiable.

The restrictions 1 and 5 ensure that all the rules for a given function demand the same extent of evaluation for a considered argument. An efficient transformation of a general BABEL program (i.e. a program possibly violating restrictions 1 and 5) to a uniform one is described in [MKLR90]. This transformation may syntactically destroy nonambiguity, but we also allow programs produced in this way, even if they do not literally satisfy restriction 6. The restrictions 4 and 6 ensure that BABEL functions are functions in the mathematical sense.

We assume some *predefined rules* for the boolean operations, guarded, and conditional expressions to be present in any program:

$\neg\ \mathit{false} := \mathit{true}.$ $\neg\ \mathit{true} := \mathit{false}.$	$X \wedge Y := \mathit{and1}\ X\ Y.$ $X \wedge Y := \mathit{and2}\ X\ Y.$ $X \wedge Y := \mathit{and3}\ X\ Y.$	$\mathit{and1}\ \mathit{false}\ Y := \mathit{false}.$ $\mathit{and2}\ X\ \mathit{false} := \mathit{false}.$ $\mathit{and3}\ \mathit{true}\ \mathit{true} := \mathit{true}.$
$X \vee Y := \mathit{or1}\ X\ Y.$ $X \vee Y := \mathit{or2}\ X\ Y.$ $X \vee Y := \mathit{or3}\ X\ Y.$	$\mathit{or1}\ \mathit{true}\ Y := \mathit{true}.$ $\mathit{or2}\ X\ \mathit{true} := \mathit{true}.$ $\mathit{or3}\ \mathit{false}\ \mathit{false} := \mathit{false}.$	$(\mathit{true} \rightarrow X) := X.$ $(\mathit{true} \rightarrow X \Box Y) := X.$ $(\mathit{false} \rightarrow X \Box Y) := Y.$

The rules given above for "$\vee$" and "$\wedge$ specify parallel disjunction and conjunction in uniform format ("parallel" as far as finite failure but no infinite computation is involved). In contrast to previous versions of BABEL [KLMR90,MKLR90], we do not assume any predefined rules for weak equality. Instead, weak equations will be reduced by constraint solving, as shown by the specification of the operational semantics in the next section.

3 Representation of Constraints: a Discussion

In a functional logic language without disequality, the only constraint-like operation is unification, which can be seen as a problem of solving sets of equations, the result being a set of bindings (equations) for variables. If one of these variables appears in a later step of the computation, its value is used instead. We claim that this nice property of having the "current constraint" as a set of pieces of information about each variable, independent of the rest, is a crucial point for an efficient implementation also in the case of disequalities.

The introduction of disequalities will require to attach more complex information to variables and to complicate the way of using it in further steps of the computation. If the language were strict, we would evaluate both sides of an equality or disequality to terms, and then we could use some of the constraint solving mechanisms proposed for the case of logic programming with disequalities [Co90,Co84,Sm91]. All of them are based on reducing a set of equalities and disequalities to some kind of *solved form*. In [Co90], solved forms are disjunctions of *basic formulas* of the form $X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge Y_1 \neq s_1 \wedge \ldots \wedge Y_m \neq s_m$, where t_i and s_j are terms, s_j is not Y_j and each X_i occurs only once. Disjunctions appear because of disequalities between constructor applications, e.g. $(c\ t_1 \ldots t_n) \neq (c\ s_1 \ldots s_n)$ is equivalent to $t_1 \neq s_1 \vee \ldots \vee t_n \neq s_n$. These solved forms are satisfiable, if all the involved variables range over infinite domains. The case of finite domains is briefly discussed later. The finiteness of a domain can be derived from the datatype definitions, even in the case of polymorphic types (see also Section 4).

Due to the laziness of our language, a disequality may hold even between infinite objects, and therefore we lose completeness, if we force the evaluation to a (finite) term. At a first sight, it seems to be enough to allow head normal forms h_j instead of terms s_j in the disequalities $Y_j \neq s_j$. An expression is in *head normal form* (HNF), if it is a variable or a constructor application $(c\ e_1 \ldots e_n)$.

Unfortunately, a conjunction $Y_1 \neq h_1 \wedge \ldots \wedge Y_m \neq h_m$ is not guaranteed to be satisfiable, as the following example shows. Let the function f be defined by the rule $f\ (\mathit{suc}\ X) := X$. Then, $Y \neq 0 \wedge Y \neq (\mathit{suc}\ (f\ Y))$ is not satisfiable, since $Y \neq (\mathit{suc}\ (f\ Y))$ is only satisfied by $Y = 0$, which contradicts $Y \neq 0$. As a conclusion, we must still require terms s_j for the disequalities $Y_j \neq s_j$. This does not mean that a head normal form h in a disequality $Y \neq h$ must be reduced to a term, which would violate the lazy nature of the language. Instead, we consider two alternatives for $Y \neq (c\ e_1 \ldots e_n)$: either $Y = (c'\ X_1 \ldots X_m)$ for some $c' \neq c$ and new variables $X_1, \ldots, X_m$ $(m \geq 0)$, or $Y = (c\ X_1 \ldots X_n)$, but for some i holds $X_i \neq e_i$. Observe that the first alternative is "lazy", that is, it does not require further evaluation of the e_i's. Furthermore, the second alternative only requires the evaluation of one selected argument.

Another question is to decide whether disjunctions within constraints in solved form will be represented explicitly or handled implicitly by the backtracking mechanism. We find two reasons against the explicit representation: Firstly, it would be difficult to maintain the representation in a "variable oriented" format (for each variable, we would have to identify and link information coming from equalities and disequalities in different disjuncts). Secondly, it would be hard to maintain incrementally the structure of solved forms (if in some disjunct D we have $X \neq s$ and an equality $X = t$ is added later on, it produces $t \neq s$ which could split into a disjunction of disequalities, and we must distribute it over the conjunction constituting D). Moreover, it is difficult, to get back the old constraint, when backtracking is needed. If we do not distribute, we get an arbitrary boolean expression, which is hard to check for satisfiability. For these reasons, we have decided to avoid explicit disjunctions. Instead, a basic formula is maintained as the current constraint within any computation state, and backtracking is used for managing disjunctions introduced by disequations between constructor applications.

Finite domains introduce a new difficulty, because a basic formula needs not be satisfiable. If, e.g., the nullary constructors a, b, c enumerate all the elements in the domain ranged over by X, then the basic formula $X \neq a \wedge X \neq b \wedge X \neq c$ is not satisfiable. The solution, we have adopted, is to generate explicit alternatives for these disequalities. In the previous example $X \neq a$ would generate (by backtracking) the alternatives $X = b$ and $X = c$. In fact, this is exactly the first alternative of the above solution. In special situations (e.g. flat finite domains), some set based representation techniques as in CHIP [DVSAGB88] could also be used.

4 Operational Semantics

We present in this section an operational semantics which is rather close to the implementation proposed in Section 5, with the exception of "sharing", which is not captured here. We give a set of rules expressing how to perform computations for evaluating a given expression. These rules constitute a modification of lazy narrowing for taking into account disequality constraints. An alternative formulation of the operational semantics could replace the explicit use of unification by means of equality constraints. This is the usual approach within the CLP scheme [JL87,HS88]. Our semantics reflects more closely the behaviour of the abstract machine, which uses (efficiently implementable (compilable)) unification instead of (interpretative) constraint solving. Our aim was to develop an implementation which behaves (as efficiently) as narrowing as long as no disequalities are involved.

The basic formulas of Section 3 are represented by two components: equalities are treated as *substitutions*, while disequalities are collected in *environments*.

A computation state, called *configuration*, is a triple $\langle e, \theta, \rho \rangle$ where

- e is the *expression* to be reduced.
- θ is a *tag*, that is, a constructor symbol c or the symbol *any*. Tags are introduced for expressing that e must be reduced to a result with top-level constructor c, or to any result in the case of *any*. Tags are interesting for avoiding in advance many useless computations by achieving a kind of "indexing by the result" mechanism and hence reducing the search space.
- ρ is an *environment*, defined as a mapping of variables to finite sets. An element of such a set can be a term or the special value *fin*.

The domain $dom(\rho)$ of an environment ρ is the set of variables X such that $\rho(X)$ is not empty. The restriction $\rho \mid_V$ of an environment ρ to a set of variables V is an environment which coincides with ρ over V and is empty for variables not in V. $fin \in \rho(X)$ indicates that X must have a finite and total value. The intended meaning of ρ is the conjunction $\bigwedge_{X \in dom(\rho)} \bigwedge_{t \in \rho(X)-\{fin\}} X \neq t$. With this reading any environment ρ is satisfiable, if no finite domain is involved. We use the notation $\rho \cup \rho'$ for the environment ρ'' with $\rho''(X) = \rho(X) \cup \rho'(X)$. For an environment ρ and a substitution σ with $dom(\rho) \cap dom(\sigma) = \emptyset$, $\rho\sigma$ denotes the result of applying σ to the elements of $\rho(X)$, for all $X \in dom(\rho)$.

Configurations are changed by the *one-step narrowing relation* $\Rightarrow_\sigma$. The corresponding rules are given below. $\langle e, \theta, \rho \rangle \Rightarrow_\sigma \langle e', \theta, \rho' \rangle$ indicates that the configuration $\langle e, \theta, \rho \rangle$ can evolve to $\langle e', \theta, \rho' \rangle$ in one step by narrowing some variables in e as specified by the idempotent substitution σ. ϵ denotes the empty substitution. The rules for $\Rightarrow_\sigma$ rely on the following auxiliary construction, used to propagate bindings through the environment.

a) Given an idempotent substitution $\sigma \equiv \{X \leftarrow t\}$ and an environment ρ, we define $propagation(\sigma, \rho) := (b, \rho')$, where b is the boolean expression

$$b := \begin{cases} t \neq t_1\sigma \wedge \ldots \wedge t \neq t_k\sigma, & \text{if } \{t_1, \ldots, t_k\} = \rho(X) - \{fin\} \\ true, & \text{if } \rho(X) - \{fin\} = \emptyset \end{cases}$$

and $\rho' := ((\rho \mid_{dom(\rho)-\{X\}})\sigma) \cup \rho''$,

where $\rho''(V) := \begin{cases} \{fin\}, & \text{if } V \in var(t) \text{ and } fin \in \rho(X) \\ \emptyset, & \text{otherwise} \end{cases}$

b) Given an idempotent substitution $\sigma \equiv \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$ and environment ρ, *propagation* $(\sigma, \rho) := (b \wedge b', \rho')$
where $(b, \rho'') := propagation(\{X_1 \leftarrow t_1\}, \rho)$
and $(b', \rho') := propagation(\{X_2 \leftarrow t_2, \ldots, X_n \leftarrow t_n\}, \rho'')$

A *computation* starting with e_0 is a sequence $\langle e_0, any, \emptyset\rangle \Rightarrow_{\sigma_1} \langle e_1, any, \rho_1\rangle \Rightarrow_{\sigma_2} \ldots \Rightarrow_{\sigma_n} \langle e_n, any, \rho_n\rangle$. A computation *succeeds* if e_n is a term and either $\theta_n = any$ or e_n is a variable or e_n is an application of the constructor θ_n. e_n is called the *result* and $\langle\sigma, \rho\rangle$ is called the *answer* of the computation, where σ is the composition $\sigma := \sigma_1 \ldots \sigma_n \mid_{var(e_0)}$, $\rho := \rho_n \mid_{descendants(var(e_0\sigma), \rho_n)}$, and $descendants(S, \rho) := S \cup descendants(var(\rho(S)), \rho)$. If a computation has not yet succeeded, but no further narrowing steps are applicable, the computation *fails*. A computation also fails, if a failure rule (Fx.y) can be applied.

Rules for $\Rightarrow_\sigma$

1. Goal expression $e \equiv (f\ e_1 \ldots e_n)$**:**

(F1.1) $\langle (f\ e_1 \ldots e_n), c, \rho\rangle \Rightarrow_\epsilon$ FAILURE

if all the rules for f *contradict* c, i.e. it is known (due to some (approximating) program analysis) for every rule for f that the rhs cannot yield a value with top-level constructor c.

(R1.1) $$\frac{\langle e_i, \theta_i, \rho\rangle \Rightarrow_\sigma \langle e_i', \theta_i, \rho'\rangle}{\langle (f\ e_1 \ldots e_i \ldots e_n), \theta, \rho\rangle \Rightarrow_\sigma \langle (f\ e_1 \ldots e_i' \ldots e_n)\sigma, \theta, \rho'\rangle}$$

if f *demands* HNF for the i-th argument, e_i is not in HNF, e_j is in HNF for every demanded $1 \leq j < i$ and θ_i is chosen as follows: if all the rules for f which do not contradict θ demand the same top level constructor c_i for the i-th argument, then θ_i is c_i; otherwise θ_i is *any*.

(R1.2) $\langle (f\ e_1 \ldots e_n), \theta, \rho\rangle \Rightarrow_\sigma \langle b \rightarrow r\lambda, \theta, \rho'\rangle$

if e_i is in HNF for every demanded $1 \leq i \leq n$, and there is a variant $(f\ t_1 \ldots t_n) := r$ (with new variables) of a rule for f, which does not contradict θ, such that

(i) $\sigma \cup \lambda$ is a most general unifier of $(f\ e_1 \ldots e_n)$ and $(f\ t_1 \ldots t_n)$, with $dom(\sigma) \subset var((f\ e_1 \ldots e_n))$ and $dom(\lambda) \subset var((f\ t_1 \ldots t_n))$. We assume that $t_i \in dom(\lambda)$ and $e_i \notin dom(\sigma)$ if both t_i and e_i, are variables.

(ii) $(b, \rho') := propagation(\sigma, \rho)$

These rules cover applications of the predefined functions $\neg$,$\vee$,$\wedge$,$\_ \ \rightarrow\ \_$ (guarded expression, if-then) and $\_ \ \rightarrow\ \_\Box\ \_$ (conditional, if-then-else). For example, in the case of the negation $\neg$, (R1.1) means that if $\langle e, false, \rho\rangle \Rightarrow_\sigma \langle e', false, \rho'\rangle$ then $\langle \neg e, true, \rho\rangle \Rightarrow_\sigma \langle \neg e', true, \rho'\rangle$

2. Goal expression $e \equiv (c\ e_1 \ldots e_n)$**:**

(F2.1) $\langle (c\ e_1 \ldots e_n), c', \rho\rangle \Rightarrow_\epsilon$ FAILURE if c' is not *any* and $c' \neq c$.

(R2.1) $$\frac{\langle e_i, any, \rho\rangle \Rightarrow_\sigma \langle e_i', any, \rho'\rangle}{\langle (c\ e_1 \ldots e_i \ldots e_n), \theta, \rho\rangle \Rightarrow_\sigma \langle (c\ e_1 \ldots e_i' \ldots e_n)\sigma, \theta, \rho'\rangle}$$

if θ is c or *any*, e_i is not a term, e_j is a term for all $1 \leq j < i$.

3. Goal expression $e \equiv X$**:**

(R3.1) $\langle X, c, \rho\rangle \Rightarrow_\sigma \langle (b \rightarrow (c\ X_1 \ldots X_n), c, \rho'\rangle$

if c is not *any*, $X_1, \ldots, X_n$ are new variables,
$\sigma := \{X \leftarrow (c\ X_1 \ldots X_n)\}$ and $(b, \rho') := propagation(\sigma, \rho)$.

4. Goal expression $e \equiv (e_1 = e_2)$ **with tag** *true***:**

(F4.1) $\langle (c\ e_1 \ldots e_n) = (c'\ e_1' \ldots e_m'), true, \rho\rangle \Rightarrow_\epsilon$ FAILURE if $c \neq c'$

(F4.2) $\langle X = t, true, \rho\rangle \Rightarrow_\epsilon$ FAILURE if t is not X and $X \in var(t)$.

(F4.2') Symmetric case of (F4.2)

(R4.1) $$\frac{\langle e_1, \theta, \rho\rangle \Rightarrow_\sigma \langle e_1', \theta, \rho'\rangle}{\langle e_1 = e_2, true, \rho\rangle \Rightarrow_\sigma \langle e_1' = e_2\sigma, true, \rho'\rangle}$$

if e_1 is not in HNF; θ is chosen as c if e_2 is in HNF with top-level constructor c, and as *any* otherwise.

(R4.2) $$\frac{\langle e_2, \theta, \rho\rangle \Rightarrow_\sigma \langle e_2', \theta, \rho'\rangle}{\langle e_1 = e_2, true, \rho\rangle \Rightarrow_\sigma \langle e_1\sigma = e_2', true, \rho'\rangle}$$

if e_1 is in HNF but e_2 is not; θ is chosen as c if e_1 is in HNF with top-level constructor c, and as *any* otherwise (i.e. e_1 is a variable).

(R4.3) $\langle (c\ e_1 \ldots e_n) = (c\ e'_1 \ldots e'_n), true, \rho\rangle \Rightarrow_\epsilon \langle (e_1 = e'_1 \wedge \ldots \wedge e_n = e'_n), true, \rho\rangle$
where $(n \geq 0)$.

(R4.4) $\langle X = (c\ e_1 \ldots e_n), true, \rho\rangle \Rightarrow_\sigma \langle (b \rightarrow (X_1 = e_1\sigma \wedge \ldots \wedge X_n = e_n\sigma), true, \rho'\rangle$
if $(c\ e_1 \ldots e_n)$ is not a term, $X_1, \ldots, X_n$ are new variables,
$\sigma := \{X \leftarrow (c\ X_1 \ldots X_n)\}$ and $(b, \rho') := propagation(\sigma, \rho \cup \{X \leftarrow \{fin\}\})$.
Note that $(c\ e_1 \ldots e_n)$ needs to be evaluated because only for finite terms, "=" can yield *true*.

(R4.4') Symmetric case of (R4.4)

(R4.5) $\langle X = X, true, \rho\rangle \Rightarrow_\epsilon \langle true, true, \rho \cup \{X \leftarrow \{fin\}\}\rangle$

(R4.6) $\langle X = t, true, \rho\rangle \Rightarrow_\sigma \langle b, true, \rho'\rangle$
if t is a term, $X \notin var(t), \sigma := \{X \leftarrow t\}$ and
$(b, \rho') := propagation(\sigma, \rho \cup \{X \leftarrow \{fin\}\})$.

(R4.6') Symmetric case of (R4.6)

5. Goal expression $e \equiv (e_1 = e_2)$ with tag *false*:

(F5.1) $\langle X = X, false, \rho\rangle \Rightarrow_\epsilon$ FAILURE

(R5.1) $\dfrac{\langle e_1, any, \rho\rangle \Rightarrow_\sigma \langle e'_1, any, \rho'\rangle}{\langle e_1 = e_2, false, \rho\rangle \Rightarrow_\sigma \langle e'_1 = e_2\sigma, false, \rho'\rangle}$
if e_1 is not in HNF.

(R5.2) $\dfrac{\langle e_2, any, \rho\rangle \Rightarrow_\sigma \langle e'_2, any, \rho'\rangle}{\langle e_1 = e_2, false, \rho\rangle \Rightarrow_\sigma \langle e_1\sigma = e'_2, false, \rho'\rangle}$
if e_1 is in HNF but e_2 is not.

(R5.3) $\langle (c\ e_1 \ldots e_n) = (c'\ e'_1 \ldots e'_m), false, \rho\rangle \Rightarrow_\epsilon \langle false, false, \rho\rangle$ if $c \neq c'$.

(R5.4) $\langle (c\ e_1 \ldots e_n) = (c\ e'_1 \ldots e'_n), false, \rho\rangle \Rightarrow_\epsilon \langle (e_1 = e'_1 \wedge \ldots \wedge e_n = e'_n), false, \rho\rangle$

(R5.5) $\langle X = (c\ e_1 \ldots e_n), false, \rho\rangle \Rightarrow_\sigma \langle b \rightarrow false, false, \rho'\rangle$
if $(c\ e_1 \ldots e_n)$ is not a term, $X_1, \ldots, X_m$ are new variables,
$\sigma := \{X \leftarrow (c'\ X_1 \ldots X_m)\}$ where c' is a constructor symbol of arity $m \geq 0$ different from c but with the same target type, and $(b, \rho') := propagation(\sigma, \rho)$.

(R5.6) $\langle X = (c\ e_1 \ldots e_n), false, \rho\rangle \Rightarrow_\sigma \langle b \rightarrow (X_1 = e_1 \wedge \ldots \wedge X_n = e_n)\sigma, false, \rho'\rangle$
if $(c\ e_1 \ldots e_n)$ is not a term, $X_1, \ldots, X_n$ are new variables,
$\sigma := \{X \leftarrow (c\ X_1 \ldots X_n)\}$, and $(b, \rho') := propagation(\sigma, \rho)$. Note that (R5.5) and (R5.6) are alternatives for the same situation, and that (R5.5) includes by itself several alternatives.

(R5.5') (R5.6') Symmetric cases of (R5.5) and (R5.6).

(R5.7) $\langle X = Y, false, \rho\rangle \Rightarrow_\epsilon \langle false, false, \rho \cup \{X \leftarrow \{Y\}, Y \leftarrow \{X\}\}\rangle$
if X, Y are different variables with a recursive or polymorphic type.

(R5.8) $\langle X = t, false, \rho\rangle \Rightarrow_\epsilon \langle false, false, \rho \cup \{X \leftarrow \{t\}\}\rangle$
if t is a non-variable term with a recursive or polymorphic type.

(R5.8') Symmetric case of (R5.8)

(R5.9) (R5.10) Like (R5.5) and (R5.6), but $(c\ e_1 \ldots e_n)$ is a term with a monomorphic non-recursive type.

(R5.11) $\langle X = Y, false, \rho\rangle \Rightarrow_\sigma \langle b \rightarrow ((c\ X_1 \ldots X_n) = Y), false, \rho'\rangle$
if X, Y are different variables with a monomorphic non-recursive type τ, c is a constructor with target type τ, $X_1, \ldots, X_n$ are new variables,
$\sigma := \{X \leftarrow (c\ X_1 \ldots X_n)\}$, and $(b, \rho') := propagation(\sigma, \rho)$.

6. Goal expression $e \equiv (e_1 = e_2)$ with tag *any*:

(R6.1) $\langle e_1 = e_2, any, \rho\rangle \Rightarrow_\epsilon \langle false = (e_1 = e_2) \rightarrow false, any, \rho\rangle$

(R6.2) $\langle e_1 = e_2, any, \rho\rangle \Rightarrow_\epsilon \langle true = (e_1 = e_2) \rightarrow true, any, \rho\rangle$

These rules have the effect of orienting $e_1 = e_2$ first to *false* and later to *true*.

The rules for disequality are similar to those for equality with the opposite tag.

The rules (R5.7) and (R5.8) handle non-recursive polymorphic types (e.g. *pair* α β) like recursive types. This is only valid, if they cannot be instantiated to a type with a finite domain (e.g. *pair bool bool*). For this reason, we assume a previous program transformation, which replaces every BABEL function by the set of its used instances (depending on the considered goal). In the *size* example, the function *size* is replaced by another function with the same rules, but type *list (pair* α *nat)* $\rightarrow$ *nat*. Note that such an instance need not be monomorphic, but it is sure that occurring type variables will not be instantiated further, especially not to a type with a finite domain. We are only interested in programs, where such a transformation is possible. We use this approach in order to avoid runtime type information, which would lead to a less efficient implementation. Note that without the mentioned program transformation non-recursive polymorphic types have to be handled like non-recursive monomorphic types, since they may be instantiated to such types.

The following example computation shows how the above rules work. Consider f defined by the BABEL rule $f\ (suc\ X)\ :=\ X$ and the configuration

(1) $\langle X = (suc\ (f\ X)),\ \ false,\ \ \{X \leftarrow \{(suc\ 0), Y, (suc\ Y)\}\}\rangle.$

By applying rule (R5.5) with $\sigma \equiv \{X \leftarrow 0\}$, we get

$$\langle 0 \neq (suc\ 0) \wedge 0 \neq Y \wedge 0 \neq (suc\ Y) \rightarrow false,\ \ false,\ \ \emptyset\rangle.$$

Using the rules for disequality corresponding to (R5.3) and (R5.8), the rules for conjunction and guarded expressions, we get $\langle false, false, \{Y \leftarrow \{0\}\}\rangle$.

Alternatively, we can apply (R5.6) with $\sigma \equiv \{X \leftarrow (suc\ Z)\}$ to (1), leading to

$$\langle (suc\ Z) \neq (suc\ 0) \wedge (suc\ Z) \neq Y \wedge (suc\ Z) \neq (suc\ Y) \rightarrow Z = (f\ (suc\ Z)), false, \emptyset\rangle.$$

Using the rules for disequality corresponding to (R5.4), (R5.8), and (R5.7), the rules for conjunction and guarded expressions, we get

$$\langle Z = (f\ (suc\ Z)),\ \ false,\ \ \{Y \leftarrow \{(suc\ Z), Z\}, Z \leftarrow \{0, Y\}\}\rangle.$$

Applying (R1.1) (for the 2nd argument of "=") and (R1.2) leads to

$$\langle Z = Z,\ \ false,\ \ \{Y \leftarrow \{(suc\ Z), Z\}, Z \leftarrow \{0, Y\}\}\rangle.$$

This computation fails since $Z = Z$ is not a term, but no rule can be applied.

4.1 The LBAM

The LBAM implements lazy narrowing for uniform BABEL programs, but it cannot represent and maintain constraints. In order to apply a function f to some arguments, the demanded arguments are evaluated to head normal form. Then, the rules for f are tried one after the other until an applicable rule is found.

Evaluating the demanded arguments before trying to apply a rule has the advantage that the arguments are only reevaluated (backtracking), if no rule is applicable. If, alternatively, the arguments are reevaluated, until the considered rule is applicable, non-termination can occur. Note that the number of rules is always finite, while the number of narrowings of an expression may be infinite (see also [JMM92]). Let for example the function *one* be defined by the two rules $one\ 0 := suc\ 0$ and $one\ (suc\ X) := one\ X$ and consider the goal $one\ (one\ Y)$. If the first rule for *one* shall be used, $(one\ Y)$ has to be narrowed to 0. Unfortunately, there are infinitely many narrowings of $(one\ Y)$, since Y can be bound to every natural number. All these narrowings deliver the (undesired) result $suc\ 0$.

The LBAM consists of:

- a *program store* containing the abstract machine code, which the BABEL program has been translated to,
- the *graph*, which contains constructor, variable, and task nodes,
- the *active task pointer* which points at the task node representing the currently executed function application.

The different kinds of nodes (see Fig. 1) are distinguished by a *tag* (first component). A *constructor node* represents a constructor application, while a *bound* or *unbound variable node* stands for a logical variable. A *task node* represents a function application. Among others, it contains the address of the code for the function, pointers to the arguments, a stack for auxiliary computations, a *status flag* (dormant, active, or evaluated), a return pointer (used to return to the activator task on successful termination), a backtracking pointer (indicating the task which must be forced to produce another solution, if the current task fails), the last descendant pointer (which points to the last (possibly indirect) descendant task, which has already finished and may produce alternative computations). This pointer is used to initialize the backtracking pointer of the next descendant task generated.

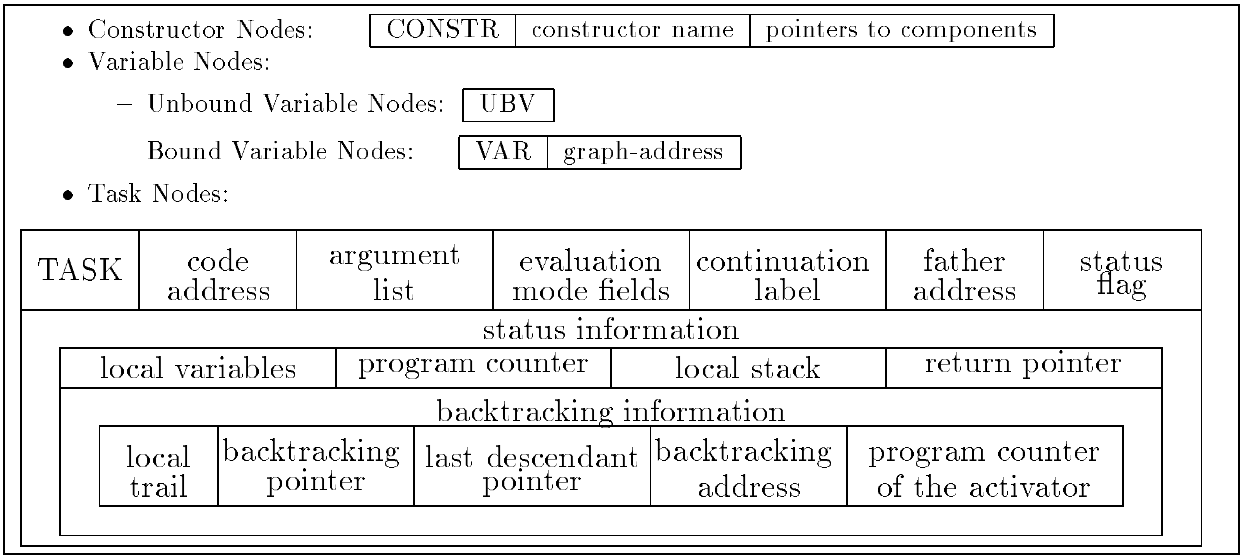

Figure 1: Structure of graph nodes.

4.2 Compiling Uniform BABEL to LBAM-Code

Each BABEL rule is translated into a sequence of small graph manipulation commands. For the compilation, the rules of a uniform BABEL program are grouped according to the function symbol they define.

The code, which is produced for a BABEL program (see Fig. 2), works as follows. First, it generates a special dormant task node for the goal (GOALNODE(goal,k_0)) (where k_0 is the number of variables in the goal), starts its evaluation (BODY_EVAL) and prints the result (PRINT_RESULT) after a successful computation. If more solutions are desired (MORE), the machine is forced to backtrack (FORCE), otherwise the program stops (JUMP_FALSE end ...STOP). After this preliminary code, the translation of the functions (using *fcttrans*) and the code for the goal follow.

The code for a BABEL function f (with k local variables) (Fig. 2 b) first evaluates (if necessary) the demanded arguments $(j_1, \ldots, j_m)$ to HNF (LOADS j_i; JUMP_HNF l_i; ARG_EVAL; l$_i$: POP ($i = 1, \ldots, m$)). This code is executed, while the father of the task for f is active, in order to place the task nodes corresponding to the demanded arguments in the backtracking chain before the task node for f.

EXECUTE activates the task for f. The rules for f are tried in their textual order. The code for a rule first sets the backtracking address l (TRY_ME_ELSE l). If the rule fails, a jump to l is performed, the bindings produced by the rule are removed (UNDO) and the next rule (if existing) is tried. If all rules fail, the predecessor of the task (usually the task for the last argument, if existing) is forced to backtrack (FAIL_RETURN).

The translation of a rule (Fig. 2 c) consists of code for the unification of the arguments of the function application with the terms on the left hand side and code for the evaluation of the expression on the right hand side.

If a term t on the left hand side is just a variable X, no code for the unification is needed, but X is used as a synonym for the corresponding argument (see Fig. 2 d). If t is a constructed term, it is unified with the corresponding argument (UNIFY).

The scheme *exptrans* (Fig. 2 e) produces code, which evaluates an expression according to the evaluation mode (HNF or NF) of the current task. First, a graphical representation G of the expression is build. If a full evaluation is needed (e.g. when the result shall be shown to the user), G is given as an argument to a special function *nfe*, predefined by the rule *nfe* X $:= X = X \rightarrow X$, and evaluated. Otherwise G is evaluated to head normal form, if still necessary.

If e has a type with a flat domain, the instructions marked with (*) in Fig. 2 can be omitted, because HNF and NF are then the same. For some expressions, e.g. the conditional and the equality, a more efficient translation is used. Due to the lack of space, we must omit this here.

The *graphtrans* (Fig. 2 f) scheme generates code which produces the graphical representation of an expression. In the case of an application, first the arguments are handled. Then, the pointers to their representations are taken from the local stack and included in a new constructor or task node respectively.

a) Code for a BABEL program:

```
     0:  GOALNODE (goal, k_0)
     1:  BODY_EVAL
     2:  PRINT_RESULT
     3:  MORE
     4:  JUMP_FALSE end
     5:  FORCE
         fcttrans (⟨f_1 t^1_{i1} ... t^1_{im_1} := e^1_i⟩^{r^1}_{i=1}, k_1)
         ...
         fcttrans (⟨f_n t^n_{i1} ... t^n_{im_n} := e^n_i⟩^{r^n}_{i=1}, k_n)
 goal:   EXECUTE
         TRY_ME_ELSE fail
         exptrans (Goal)
 fail:   PRINT_FAILURE
 end:    STOP.
```

b) *fcttrans* $(\langle f\ t_{i1} \ldots t_{im} := e_i\rangle^r_{i=1},\ k)$:=

```
 f:       LOADS j_1
          JUMP_HNF l_1
          ARG_EVAL
 l_1:     POP
          ...
          LOADS j_k
          JUMP_HNF l_k
          ARG_EVAL
 l_k:     POP
          EXECUTE
          TRY_ME_ELSE rule_2
          ruletrans (f t_11 ... t_1m := e_1)
 rule_2:  UNDO
          TRY_ME_ELSE rule_3
          ruletrans (f t_21 ... t_2m := e_2)
 rule_3:  UNDO
          ...
 rule_r:  UNDO
          TRY_ME_ELSE l_fail
          ruletrans (f t_r1 ... t_rm := e_r)
 l_fail:  UNDO
          FAIL_RETURN
```

c) *ruletrans* $(f\ t_1 \ldots t_m := e)$:=

```
          LOAD 1
          unifytrans (t_1)
          ...
          LOAD m
          unifytrans (t_m)
 explb:   exptrans (e)
          RETURN
```

d) *unifytrans* (X) := POP

```
 unifytrans(c X_i ... X_{n+i-1}) :=
     UNIFY (c,n,i)
```

e) *exptrans* (e) :=

```
          graphtrans (e)
          JUMP_EMODE l_hnf          (*)
          NODE (nfe, 1, 1)          (*)
          BODY_EVAL                 (*)
          JUMP l_end                (*)
 l_hnf :  JUMP_HNF l_end
          BODY_EVAL
 l_end :  ...
```

f) *graphtrans* (X_i) := LOADX i

```
 graphtrans (X) := LOAD i
   if X is a synonym for argument i
 graphtrans ((c e_1 ... e_n)) :=
     graphtrans (e_1)
     ...
     graphtrans (e_n)
     CNODE (c, n)
 graphtrans ((f e_1 ... e_n)) :=
     graphtrans (e_1)
     ...
     graphtrans (e_n)
     NODE (f, n, k)
```

Figure 2: Code Generation Schemes

4.3 Extensions of the LBAM to Cope with Constraints

BABEL allows equality and disequality constraints. In Section 3, we have seen that it is sufficient to deal with basic formulas, i.e. constraints, which are the conjunction of *elementary constraints*, where the left hand side is always a variable.

If there is an equality constraint for a variable X, it will be the only constraint concerning this variable. Hence, this constraint can be handled by binding X to the corresponding right hand side. But in contrast to a binding during unification, we now have to store that the variable has to be finite, since equality is only defined for finite values. Hence, we need an additional tag in each bound variable node, indicating whether the variable may only be bound to finite values. This information will not be used during computation, since it is undecidable, whether some computation is going to be finite, but it will be given to the user at the end.

If there is no equality constraint for some variable X, there may be a conjunction of elementary disequality constraints for X. To handle this, we introduce a new kind of node called *constraint node*. The variable node for X will point to such a node. A constraint node contains the tag CONSTRAINT and a list of pointers, each pointing to the graphical representation of a term t (representing the constraint X$\neq t$).

Note that a variable node is not directly overwritten by a constraint node, but a (bound) variable node will point to it. The reason for this is that a constraint is usually changed several times during computation. While backtracking, the old constraint has to be reestablished. This is much easier, if the pointers to all the constraint nodes, representing "old" constraints, are trailed.

In addition to a new kind of node, we need a new notion of *head normal form*. Now, an expression is in HNF, if it is an unbound variable, a constructor application or a *constrained variable* (represented by a variable node pointing to a constraint node).

Furthermore, it is more complicated to decide whether a rule is applicable (see Section 4). More constraints may have to be added in order to apply a rule. It may be the case, that an actual argument is a constrained variable X which has to be unequal to a list of terms *tlist*, while the corresponding formal argument is a term $t := (c\ \mathrm{X}_1 \ldots \mathrm{X}_n)$. Suppose that *tlist* contains a term $t' := (c\ t_1 \ldots t_n)$. In order to apply the rule, we have to bind X to t. From the old constraint X$\neq t'$, we now get the new constraint $t \neq t'$. In order to fulfill $t \neq t'$, we have to select an argument position i $(1 \leq i \leq n)$ and to add the constraint $\mathrm{X}_i \neq t_i$. If the selected i leads to a failure later on, we have to be able to modify this selection. One possible approach is to use a second backtracking mechanism for this (besides trying the next rule, if the considered rule leads to a failure).

In order to avoid a very complicated and probably inefficient backtracking scheme, we will use the current backtracking mechanism to handle this second source of backtracking. This constitutes a valuable simplification of the implementation. The idea is that the new constraint, which is generated while "unifying" a formal argument with a constrained variable (as the actual argument), is handled as if it would be part of the guard of the corresponding rule (see rule (R1.2) in Section 4).

```
         GOALNODE(goal,k0)
         BODY_EVAL
         PRINT_RESULT
         MORE
         JUMP_FALSE end
         FORCE
member:  LOADS 2
         JUMP_HNF l1
         ARG_EVAL
   l_1:  POP
         EXECUTE
         TRY_ME_ELSE rule2
         LOAD 2
         UNIFY (nil,0,0)
         INCLUDE_CSTRT
         CNODE (false,0)
         RETURN
 rule2:  UNDO
         TRY_ME_ELSE fail
         LOAD 2
         UNIFY (cons,2,1)
         INCLUDE_CSTRT
         LOAD 1
         LOADX 1
         NODE(=,2,0)
         BODY_EVAL
         JUMP_FALSE else
         CNODE (true,0)
         JUMP ret
  else:  LOAD 1
         LOADX 2
         NODE(member,2,2)
         JUMP_EMODE hnf
    nf:  NODE(nfe,1,1)
         BODY_EVAL
         JUMP ret
   hnf:  JUMP_HNF ret
         BODY_EVAL
   ret:  RETURN
  fail:  UNDO
         FAIL_RETURN
  size:  ...(similarly) ...
  goal:  EXECUTE
         TRY_ME_ELSE exit
         LOADX 1
         CNODE (mkpair,2)
         LOADX 2
         LOADX 3
         CNODE (mkpair,2)
         CNODE (nil,0)
         CNODE (cons,2)
         CNODE (cons,2)
         NODE(size,2,2)
         JUMP nf
              (* simplified *)
  exit:  PRINT_FAILURE
   end:  STOP
```

Figure 3: CBAM code for the size example

Technically, this is accomplished by extending each task node by a pointer to a graphical representation of the boolean expression representing the constraint *cstr* accumulated up to now, that means while unifying the previous arguments with the corresponding formal arguments of the rule (*cstr* initially points to a node for "true"). This pointer will be redirected to the graphical representation of $t \neq t'_1 \wedge \ldots \wedge t \neq t'_k \wedge cstr$ (where $t'_1, \ldots, t'_k$ are all the terms in *tlist*) by the UNIFY command, which is responsible for the unification with t. After the unification phase and before the evaluation of the right hand side this boolean expression is pushed onto the stack and evaluated. This is done by a new command INCLUDE_CSTRT, which is inserted before label *explb* in the *ruletrans* scheme. Using this idea, constraint propagation is transformed into the evaluation of boolean expressions. The code for the example of Subsection 2.1 is shown in Figure 3.

4.4 Result Oriented Computing

The attentive reader may have observed that the destination oriented computing ("tag " mechanism) proposed in Section 4 has not been implemented in our abstract machine. In fact, this is not needed, since an equivalent effect can be obtained through a simple program transformation. This transformation is described in detail in [KL92]. Due to lack of space, we only sketch it here. A rule $f\ t_1\ \ldots\ t_n := e$ is replaced by the sequence of rules $f\ t_1\ \ldots\ t_n\ d_{c_i} := e'_i \quad (1 \leq i \leq m)$ if e may produce $c \in \{c_1, \ldots, c_m\}$ as the outermost constructor, and where $d_{c_1}, \ldots, d_{c_m}$ are new nullary constructors. Moreover, each application $(f\ e_1\ \ldots\ e_n)$ is replaced by $(f\ e_1\ \ldots\ e_n\ d)$, where d is the outermost constructor of the desired result. e'_i is the result of transforming e to receive the desired outermost constructor c_i ($1 \leq i \leq n$). If arbitrary solutions are allowed, d is a new unbound variable.

5 Conclusions and Future Work

We have presented a variant of the lazy functional logic language BABEL [MR88] [MR92] which incorporates disequality constraints for solving equations and building answers. This enhances the computational power of the language, since disequalities as answers may replace infinitely many answer substitutions. We have developed an operational semantics which combines lazy narrowing with disequality constraint solving. Also a useful optimization, "result oriented computation", has been integrated in a clean way into the operational semantics.

For the implementation of the language, we have shown how the narrowing machine LBAM [MKLR90] can be extended to cope with disequality constraints. The mechanisms are rather independent of the LBAM and can be inserted into other narrowing machines, e.g. the stack based narrowing machine from [Lo91], as well. If there are no disequalities in the program, the machine behaves exactly like the LBAM, and no additional overhead is needed.

We are currently working on the implementation of the presented machine and hope to have a running prototype soon. In the future, we want to extend BABEL also by other kinds of constraints.

Acknowledgements

We thank Ana Gil for many helpful discussions.

References

[BBCMMS89] G.P. Balboni, P.G. Bosco, C. Cecchi, R. Melen, C. Moiso, G. Sofi: Implementation of a Parallel Logic Plus Functional Language, in: P. Treleaven (ed.), *Parallel Computers: Object Oriented, Functional and Logic*, Wiley'89.

[BCGMP89] P.G. Bosco, C. Cecchi, E. Giovannetti, C. Moiso, C. Palamidessi: Using Resolution for a Sound and Efficient Integration of Logic and Functional Programming, in: J. de Bakker (ed.), *Languages for parallel architectures: Design, Semantics, Implementation Models*, Wiley, 1989.

[BL86] M. Bellia, G. Levi: The Relation between Logic and Functional Languages, Journal of Logic Programming, Vol.3, 1986, 217-236.

[CL89] H. Comon, P. Lescanne: Equational problems and disunification. J. of Symbolic Computation, 7, 1989, 371-425.

[Co82] A. Colmerauer: Prolog and infinite trees, in K.L. Clark, S.A. Tarnlund (eds.) Logic Programming, Academic Press, 1982, 231-251.

[Co84] A. Colmerauer: Equations and inequations on finite and infinite trees, Procs. FGCS'84, 1984, 85-99.

[Co90] H. Comon: Unification: A Survey, Tech. Rep. 540, LRI, Orsay, 1990.

[DL86] D. DeGroot, G. Lindstrom (eds.): *Logic Programming: Functions, Relations, Equations*, Prentice Hall, 1986.

[DVSAGB88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graft, F. Bertheir: The constraint logic programming CHIP, Procs. Int. Conf. 5th Generation Computer Systems, FGCS'88, 1988, 693-702.

[Ha90] M. Hanus: Compiling Logic Programs with Equality, Workshop on Progr. Language Impl. and Logic Progr. (PLILP), LNCS 456, 1990, 387–401.

[HS88] M. Höhfeld, G. Smolka: Definite Relations over Constraint Languages, LILOG Report 53, IBM Germany, 1988 (to appear in J. of Logic Progr.).

[JL87] J. Jaffar, J.L. Lassez: Constraint Logic Programming, Procs. 14th ACM Symp. on Princ. of Prog. Lang., 1987, 114-119.

[JMM92] J.A. Jiménez-Martín, J. Mariño-Carballo, J.J. Moreno-Navarro: Efficient Compilation of Lazy Narrowing into Prolog, to appear in: Procs. LOPSTR'92, Springer, 1992.

[KL92] H. Kuchen, F.J. López-Fraguas: Result Directed Computing in a Functional Logic Language, Tech. Report, RWTH Aachen, 1992.

[KLMR90] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-based Implementation of a Functional Logic Language, ESOP, LNCS 432, 1990, 271-290.

[KLMR92] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Graph-Narrowing to Implement a Functional Logic Language, Tech. Report, Univ. Politecnica Madrid, 1992.

[KLMR92b] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Implementing Disequality in a Functional Logic Language, to appear in: Procs. Symp. on Logic Programming, MIT Press, 1992.

[Lo91] R. Loogen: From Reduction Machines to Narrowing Machines, TAPSOFT'91, LNCS 494, 438-457.

[Lo92] F.J. López-Fraguas: A General Scheme for Constraint Functional Logic Programming, to appear in Procs. ALP'92, LNCS.

[LR91] F.J. López-Fraguas, M. Rodríguez-Artalejo: An Approach to Constraint Functional Logic Programming, Tech. Rep. DIA 91/4, 1991.

[Mi78] R. Milner: A Theory of Type Polymorphism in Programming, JCSS 17(3), 1978, 348-375.

[MKLR90] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, M. Rodríguez-Artalejo: Lazy Narrowing in a Graph Machine, ALP, LNCS 463, 1990, 298-317; detailed version appeared as: Aachener Informatik-Bericht Nr. 90-11.

[MR88] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: BABEL: A functional and logic language based and constructor discipline and narrowing, Procs. 1st Int. Conf. on Algebraic and Logic Progr. (ALP), LNCS 343, 1989, 223-232.

[MR92] J.J. Moreno-Navarro, M. Rodríguez-Artalejo: Logic Programming with Functions and Predicates: The Language BABEL, J. Logic Programming, 12, 1992, 189-223.

[Mu90] A. Mück: Compilation of Narrowing, PLILP'90, LNCS 456, 16-39 (1990).

[Re85] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages, Procs. Int. Symp. on Logic Programming, 1985, 138-151.

[Sm91] D.A. Smith: Constraint Operations for CLP(FT), Procs. 8th Int. Conf. on Logic Programming, MIT Press, 1991, 760-774.

Formal Specification of the Abstract Machine

A Basic Components of the Machine

A.1 Data Area

1. Graph := Graph_Address $- \rightarrow$ Nodes
   (a) Graph_Address := $\mathbb{N}$
   (b) Nodes := Terminal_Nodes $\cup$ Variable_Nodes $\cup$ Task_Nodes
2. Terminal_Nodes :=

$$\begin{array}{l} \{\ \textsc{constr}\ \} \times \{\ c \mid c \in \text{Constructors},\ \text{arity}(c) \geq 0\ \} \times \text{Graph\_Address}^* \\ \cup\ \{\ \textsc{constraint}\ \} \times \text{Graph\_Address}^* \end{array}$$

3. Variable_Nodes

$$\begin{array}{llll} & := & \{\ \textsc{ubv}\ \} & \{\text{unbound variables}\} \\ & \cup & \{\ \textsc{var}\ \} \times \text{Graph\_Address} \times \{\ \textsc{fin},\ \textsc{inf}\ \} & \{\text{bound variables}\} \end{array}$$

4. Task_Nodes

$$\begin{array}{lll} := & \{\ \textsc{task}\ \} & \\ \times & \text{Program\_Address} & \{\text{code address}\} \\ \times & \text{Graph\_Address}^* & \{\text{argument addresses}\} \\ \times & \{\ \textsc{hnf},\ \textsc{nf}\ \} & \\ \times & \{\ \textsc{bdev},\ \textsc{argev}\ \} & \{\text{evaluation modes}\} \\ \times & \text{Graph\_Address} & \{\text{constraint}\} \\ \times & \text{Program\_Address} & \{\text{continuation address}\} \\ \times & \text{Graph\_Address} & \{\text{father pointer}\} \\ \times & \text{Status\_Information} & \end{array}$$

   (a) Status_Information

$$\begin{array}{ll} := & \text{Dormant\_Task\_Information} \\ \cup & \text{Ready\_Task\_Information} \\ \cup & \text{Active\_Task\_Information} \end{array}$$

   (b) Dormant_Task_Information := $\{\ \textsc{dormant}\ \} \times \mathbb{N}$ $\{\text{number of local variables}\}$

   (c) Ready_Task_Information

$$\begin{array}{llll} := \{\ \textsc{ready}\ \} & \times & \mathbb{N} & \{\text{number of local variables}\} \\ & \times & \text{Graph\_Address} & \{\text{result of the task}\} \end{array}$$

   (d) Active_Task_Information

$$\begin{array}{lll} := & \{\ \textsc{active},\ \textsc{evaluated}\ \} & \\ \times & \text{Graph\_Address}^* & \{\text{local variables}\} \\ \times & \text{Program\_Address} & \{\text{program counter}\} \\ \times & \text{Graph\_Address}^* & \{\text{local stack}\} \\ \times & \text{Graph\_Address} & \{\text{return pointer}\} \\ \times & \text{Backtracking\_Information} & \end{array}$$

   (e) Backtracking_Information :=

$$\begin{array}{ll} (\text{Graph\_Address} \cup & \\ \text{Graph\_Address}^2 \times \{\ \textsc{fin},\ \textsc{inf}\ \})^* & \{\text{local trail}\} \\ \times\ \text{Graph\_Address} & \{\text{backtracking pointer}\} \\ \times\ \text{Graph\_Address} & \{\text{last descendant pointer}\} \\ \times\ \text{Program\_Address} & \{\text{backtracking address}\} \\ \times\ \text{Program\_Address} & \{\text{copy of father's program counter}\} \end{array}$$

A.2 Instructions

$$\begin{array}{rcl} \text{Instructions} & := & \text{Local\_Stack\_Instructions} \\ & \cup & \text{Graph\_Instructions} \\ & \cup & \text{Unification\_Instructions} \\ & \cup & \text{Jump\_Instructions} \\ & \cup & \text{Process\_Instructions} \\ & \cup & \text{Basic\_Operations} \\ & \cup & \text{Constraint\_Instructions} \\ & \cup & \text{Additional\_Instructions} \end{array}$$

1. Local_Stack_Instructions := { POP },
2. Graph_Instructions :=
   { LOAD i, LOADX i, LOADS i, LOADARGS | i $\in \mathbb{N}$ },
   $\cup$ { CONSTRNODE (c, arity) | c $\in$ Constructors, arity $\in \mathbb{N}$ }
   $\cup$ { NODE (ca, arity, locals) | ca $\in$ Program_Address, arity, locals $\in \mathbb{N}$}
   $\cup$ { GOALNODE (ca, locals) | ca $\in$ Program_Address, locals $\in \mathbb{N}$}
3. Unification_Instructions :=
   { UNIFY (c, arity, args) | c $\in$ Constructors, arity, args $\in \mathbb{N}$}
4. Jump_Instructions :=

   { JUMP lb, JUMP_TRUE lb, JUMP_FALSE lb, JPTNP lb, JPFNP lb, JUMP_HNF lb,
   JUMP_EVAL_MODE lb, JPNONVAR lb, JPES lb | lb $\in$ Program_Address }

5. Process_Instructions :=

   { TRY_ME_ELSE lb | lb $\in$ Program_Address }
   $\cup$ { ARG_EVALUATE, EVALUATE, EXECUTE, RETURN, LASTRETURN,
   FAIL_RETURN, UNDO, FORCE }

6. Constraint_Instructions := { INCLUDE_CONSTRAINT}
7. Basic_Operations := { CHECKEQ, CHECKDISEQ }
8. Additional_Instructions := { STOP, PRINT_RESULT, PRINT_FAILURE, MORE }

A.3 States of the Machine

Machine_State := ProgramStore $\times$ Graph_Address $\times$ Graph where

(a) Program_Store := Program_Address $\rightarrow$ Instructions

(b) Program_Address := $\mathbb{N}$

Initial Configuration: (Pr, atp_0, G_0) where

- Pr $\in$ Program_Store,
- $\text{atp}_0 \in$ Graph_Address,
- $G_0(\text{atp}_0) = (\textsc{task}, 0, \varepsilon, \textsc{nf}, \textsc{bdev}, -, -, -, \textsc{active}, \varepsilon, 0, \varepsilon, -, (\varepsilon, -, \text{atp}_0, -, -))$

B Translation of Uniform Babel Programs

Babel Program := FCT (f_1, m_1, k_1, D(f_1))

$\vdots$

FCT (f_n, m_n, k_n, D(f_n))
GOAL (k_0)

Code for a Babel Program:

```
          0:  GOALNODE (goal, k_0)
          1:  EVALUATE
          2:  PRINT_RESULT
          3:  MORE
          4:  JUMP_FALSE end
   force_lb:  FORCE
              fcttrans (FCT (f_1, m_1, k_1, D(f_1)))
                ⋮
              fcttrans (FCT (f_n, m_n, k_n, D(f_n)))
       goal:  EXECUTE
              TRY_ME_ELSE last_fail
              exptrans (GOAL (k_0))
              RETURN
  last_fail:  PRINT_FAILURE
        end:  STOP
```

B.1 Translation of a Function

fcttrans : Function $\rightarrow$ Code

Let FCT (f, m, k, D(f)) = { f $t_{i,1} \ldots t_{i,m}$:= $\mathrm{body}_i(X_1,\ldots,X_k)$ | $1 \leq i \leq n$, $\forall$ j $\in$ D(f): $t_{i,j} \notin \mathit{Var}$ } where D(f) denotes the set of demanded arguments, i.e. arguments where all the rules have constructed terms in the corresponding positions on the left hand sides.

fcttrans (FCT (f, m, k, $\{j_1,\ldots,j_r\}$)) :=

```
      ca(f):  LOADS j_1
              JUMP_HNF l_1
              ARG_EVALUATE
        l_1:  POP
              LOADS j_2
                ⋮
              LOADS j_r
              JUMP_HNF l_r
              ARG_EVALUATE
        l_r:  POP
              EXECUTE
              TRY_ME_ELSE label_1
              ruletrans (f t_{1,1} ... t_{1,m} := body_1(X_1,...,X_k))
              RETURN
    label_1:  UNDO
              TRY_ME_ELSE label_2
              ruletrans (f t_{2,1} ... t_{2,m} := body_2(X_1,...,X_k))
              RETURN
    label_2:  UNDO
                ⋮
  label_n-1:  UNDO
              TRY_ME_ELSE label_n
              ruletrans (f t_{n,1} ... t_{n,m} := body_n(X_1,...,X_k))
              LASTRETURN
    label_n:  UNDO
              FAIL_RETURN
```

The demanded arguments $j_1,\ldots,j_r$ are evaluated under the control of the father (see the semantics of the commands LOADS, ARG_EVALUATE, and EVALUATE) in order to get the backtracking chain right: only if all the rules for the considered function fail, backtracking to the arguments should occur. When all demanded arguments are successfully evaluated, control is given to the considered function by the EXECUTE command. This approach was chosen in order to avoid infinite computations, if infinitely many undesired narrowings of an argument are possible (see the one example on page 8).

B.2 Translation of a Rule

ruletrans: Rules → Code

ruletrans (f $t_1 \dots t_m$:= body) := LOAD 1
unifytrans (t_1)
LOAD 2
unifytrans (t_2)
⋮
LOAD m
unifytrans (t_m)
INCLUDE_CONSTRAINT
POP
exptrans (body)

ruletrans uses the scheme *unifytrans* to unify an argument of a function application with the corresponding simple term on the left hand side of a Uniform Babel rule.

unifytrans: Term → Code

unifytrans (X) := POP

X is taken as a synonym for the corresponding argument. Hence, nothing needs to be done.

unifytrans (c $X_i \dots X_{i+n-1}$) := UNIFY (c, n, i)

Remember that in uniform Babel programs, we only have simple terms of this kind on the left hand side of a rule. Nested constructor applications cannot occur.

B.3 Translation of an Expression

For the translation of an expression, we supply two schemes: *graphtrans* produces code, which generates a graphical representation of an expression without evaluating it, and *exptrans* produces code which evaluates an expression to head normal form or normal form depending on the current evaluation mode.

graphtrans: Exp → Code

graphtrans (X) := LOAD i, if X is a synonym for argument i

graphtrans (X_i) := LOADX i, if X_i is a local variable

graphtrans (c $e_1 \dots e_n$) := *graphtrans* (e_1)
⋮
graphtrans (e_n)
CONSTRNODE (c, n)

graphtrans (f $e_1 \dots e_n$) := *graphtrans* (e_1)
⋮
graphtrans (e_n)
NODE (ca(f), n, k)

Here, guarded and conditional expressions are handled as applications of the predefined functions $\_ \rightarrow \_$ and $\_ \rightarrow \_\Box\_$. The last definition also covers applications of the predefined operations like $\vee$, $\wedge$, $=$, $\neq$, and $\neg$.

exptrans: Exp → Code

exptrans (t) := *graphtrans* (t) if $t \in \mathit{Term} - \mathit{Var}$

exptrans (e) := *graphtrans* (e) if (e = c $e_1 \dots e_n$ and $e \notin \mathit{Term}$) or e = f $e_1 \dots e_n$ or $e \in \mathit{Var}$
JUMP_EVAL_MODE hnf_lb
NODE (ca(nfe), 1, 1)
EVALUATE

```
           JUMP end_lb
hnf_lb:    JUMP_HNF end_lb
           EVALUATE
end_lb:    ...
```

nfe (normal form evaluation) is a predefined function shown later. If e has a flat type (e.g. enumeration type, bool, ...), this can be simplified to

exptrans (e) := *graphtrans* (e)
JUMP_HNF end_lb
EVALUATE
end_lb: ...

This definition covers applications of the predefined operations $\vee$, $\wedge$, $=$, $\neq$, and $\neg$. The JUMP_HNF instruction can always be omitted, if it known that the top of stack is not in HNF.

exptrans ($b \rightarrow e_1 \Box e_2$) := *exptrans*($b$)
JUMP_FALSE l_1
exptrans(e_1)
JUMP l_2
l_1: *exptrans*(e_2)
l_2: ...

exptrans ($b \rightarrow e$) := *exptrans*(b)
JUMP_FALSE force_lb
exptrans(e)

The disadvantage of this simple approach is that b might be evaluated to *false*, which causes immediate backtracking. A more sophisticated scheme which only evaluates b to *true* is shown in Appendix E. force_lb (here and always) refers to the label of the unique FORCE instruction (see page 16).

C Behaviour of the Machine

$\vdash$: Machine_State $-\rightarrow$ Machine_State

(Pr, atp, G) $\vdash$ (Pr, atp$'$, G$'$) <u>if</u> G (atp) = (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)
<u>and</u> Pr (pc) is defined
<u>and</u> SI [[Pr (pc)]] (atp, G) = (atp$'$, G$'$)

where the semantics of the instructions (semantic function)

SI : Instructions $\times$ Graph_Address $\times$ Graph $\rightarrow$ Graph_Address $\times$ Graph

is defined as shown below. First, we will define some auxiliary functions. *dereference* follows a chain of pointers, until an "interesting" node is found:

dereference : Graph $\times$ Graph_Address $\rightarrow$ Graph_Address

dereference (G, adr) :=
<u>if</u> G (adr) = (VAR, adr$'$, fi) <u>or</u>
G (adr) = (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, EVALUATED, locvl, pc, st:adr$'$, rptr, binf) <u>or</u>
G (adr) = (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, READY, k, adr$'$)
<u>then</u> *dereference* (G, adr$'$) <u>else</u> adr

new determines free nodes in the Graph:

new : Graph $\times \mathbb{N} \rightarrow$ Graph_Address*

new(G, k) := a_1:...:a_k
with $a_0 := 0$
$a_i := \min\{a \mid G(a)$ is undefined and $a > a_{i-1}\}$ for $i > 0$

Moreover, we use *term*(G,v) to determine, whether the node at address v in graph G is the root of a tree (or dag) representing a term, where

term: Graph $\times$ Graph_Address $\rightarrow$ Bool

term(G,v) := $\underline{\text{let}}$ v$'$:= *dereference*(G,v) $\underline{\text{in}}$
$\underline{\text{if}}$ G(v) = (UBV) $\underline{\text{or}}$ G(v) = (VAR, ptr, fi) $\underline{\text{then}}$ *true*
$\underline{\text{else}}$ $\underline{\text{if}}$ G(v) = (TASK, ...) $\underline{\text{then}}$ *false*
$\underline{\text{else}}$ $\underline{\text{let}}$ (CONSTR, c, p_1:...:p_n) := G(v) $\underline{\text{in}}$
term(G,p_1) $\underline{\text{and}}$ ... $\underline{\text{and}}$ *term*(G,p_n)

build(G,c,cstrt,$p_1 : \ldots : p_n, p$) produces a pair (cstrt$'$,G$'$), where cstrt$'$ is the old constraint cstrt extended by (a graphical representation of) $e(p) \neq e(p_1) \wedge \ldots \wedge e(p) \neq e(p_n)$. Here, $e(ptr)$ denotes the expression e (possibly a term) where ptr points to the root of the graphical representation of e in G. G$'$ is the graph G extended by this graphical representation.

build: Graph $\times$ Constructor $\times$ Graph_Address $\times$ Graph_Address* $\times$ Graph_Address $\rightarrow$
Graph_Address $\times$ Graph

build(G, c, cstrt, ε, adr) := (cstrt, G)

build(G, c, cstrt, cstr_1:rest, adr) :=
$\underline{\text{if}}$ G(cstr_1) = (CONSTR, c', args) $\underline{\text{and}}$ $c \neq c'$
$\underline{\text{then}}$ *build*(G, c, cstrt, rest, adr)
$\underline{\text{else}}$ { G(cstr_1) = (UBV) $\underline{\text{or}}$ G(cstr_1) = (CONSTR, c, args) }
$\underline{\text{if}}$ G(cstrt) = (TASK, ca(and), argl, ...) $\underline{\text{then}}$
build(G[cstrt | (TASK, ca(and), argl:*new*(G,1), –, –, –, –, –, DORMANT, 0),
new(G,1) | (TASK, ca(neq), cstr_1:adr, –, –, –, –, –, DORMANT, 0),
c, cstrt, rest, adr)
$\underline{\text{else}}$ { cstrt = – }
$\underline{\text{let}}$ cstrt$'$:new := *new*(G,2) $\underline{\text{in}}$
build(G[cstrt$'$ | (TASK, ca(and), new, –, –, –, –, –, DORMANT, 0),
new | (TASK, ca(neq), cstr_1:adr, –, –, –, –, –, DORMANT, 0),
c, cstrt$'$, rest, adr)

The predefined operations including "and" and "neq" will be shown in Appendix D.

In order to improve the readability of the definitions, the modified parts of the state are emphasized (e.g. *pc+1*). Constants (tags) are denoted in small capitals (e.g. TASK, VAR, ACTIVE, BDEV).

Many commands (as well as the above auxiliary definitions) make assumptions (formulated as $\underline{\text{let}}$-clauses) about the situation in which they are executed. The code generation schemes ensure that these assumptions are right. In other situations, the commands are just undefined.

C.1 Semantics of Local Stack Instructions

SI ⟦ POP ⟧ { pop top of stack }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *pc+1, st*, rptr, binf)])

C.2 Semantics of Graph Instructions

SI ⟦LOAD i⟧ { load argument i to the stack }
(atp, G [atp | (TASK, ca, arg_1:...:arg_i:...:arg_m, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st, rptr, binf)])
:= (atp, G [atp | (TASK, ca, arg_1:...:arg_i:...:arg_m, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1, st:arg*$_i$, rptr, binf)])

SI ⟦LOADX i⟧ { load local variable i to the stack }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locv_1:...:locv_i:...:locv_k, pc, st, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locv_1:...:locv_i:...:locv_k, *pc+1, st:locv*$_i$, rptr, binf)])

SI ⟦LOADS i⟧ { load argument i of son to the stack }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= let (TASK, ca′, arg_1:…:arg_i:…:arg_m, evm′, cevm′, cstrt′, rtlb′, fptr′, DORMANT, k) :=
G(*dereference*(G,adr))
in (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, *st:adr:*arg_i, rptr, binf)])

SI ⟦CONSTRNODE (c, m)⟧ { generate a constructor node }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:a_1:…:a_m, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, *st:new(G,1)*, rptr, binf),
new(G,1) | (CONSTR, c, a_1:…:a_m)])

SI ⟦NODE (caf, m, k)⟧ { generate a node for a function application }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:a_1:…:a_m, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, *st:new(G,1)*, rptr, binf),
new(G,1) | (TASK, caf, a_1:…:a_m, –, – ,– , –, –, DORMANT, k)])

SI ⟦GOALNODE (caf, k)⟧ { generate a node for the goal }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, *new(G,1)*,
ACTIVE, locvl, *pc+1*, *st:new(G,1)*, rptr, binf),
new(G,1) | (TASK, caf, ε, –, –, –, –, –, DORMANT, k)])

C.3 Semantics of Unification Instructions

SI ⟦UNIFY (c, m, l)⟧ { unify top of stack with constructor term }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, $locv_1$:…:$locv_k$, pc, st:adr, rptr, (tr, bptr, ldpt, bad, spc))])
:= let adr′ = *dereference*(G,adr) in
if G (adr′) = (CONSTR, c′, arg_1 :…:arg_n)
then if c = c′ and m = n
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, $locv_1$:…:$locv_k$, *pc+1*, *st*, rptr,
(*tr:*$locv_l$:…:$locv_{l+m-1}$, bptr, ldpt, bad, spc)),
$locv_{i+l-1}$ | (VAR, arg_i, INF) i=1,…,m])
else {c ≠ c′ or m ≠ n, unification fails}
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, $locv_1$:…:$locv_k$, *bad*, *st*, rptr, (tr, bptr, ldpt, bad, spc))])
else if G (adr′) = (UBV)
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, $locv_1$:…:$locv_k$, *pc+1*, *st*, rptr,
(*tr:adr′*, bptr, ldpt, bad, spc)),
adr′ | (CONSTR, *c*, $locv_l$:…:$locv_{l+m-1}$)])
else { G(adr′ = (VAR, ptr, fi) and G(ptr) = (CONSTRAINT, cstrts) }
(atp, G′[atp | (TASK, ca, argl, evm, cevm, *cstrt′*, rtlb, fptr,
ACTIVE, $locv_1$:…:$locv_k$, *pc+1*, *st*, rptr,
(*tr:(adr′,ptr, fi)*, bptr, ldpt, bad, spc)),
adr′ | (CONSTR, *c*, $locv_l$:…:$locv_{l+m-1}$)])
where (cstrt′, G′) := *build*(G, c, cstrt, cstrts, adr′)

C.4 Semantics of Jump Instructions

SI ⟦JUMP lb⟧ { unconditional jump }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb*, st, rptr, binf)])

SI ⟦JUMP_TRUE lb⟧ { jump if top of stack *true* }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])

:= let (CONSTR, bval, ε) := G(*dereference*(adr)) in
if bval = *true*
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb, st*, rptr, binf)])
else { bval = *false* }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *pc+1, st*, rptr, binf)])

SI ⟦JUMP_FALSE lb⟧ { jump if top of stack *false* }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= let (CONSTR, bval, ε) := G (*dereference*(adr)) in
if bval = *false*
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb, st*, rptr, binf)])
else { bval = *true* }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *pc+1, st*, rptr, binf)])

SI ⟦JPFNP lb⟧ { jump if top of stack *false*, no pop }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= let (CONSTR, bval, ε) := G(*dereference*(adr)) in
if bval = *false*
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb, st*, rptr, binf)])
else { bval = *true* }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st:adr, rptr, binf)])

JPTNP analogously.

SI ⟦JUMP_HNF lb⟧ { jump if top of stack in HNF }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= if G (*dereference*(adr)) = (CONSTR, c, argl′) or G (*dereference*(adr)) = (UBV) or
(G (*dereference*(adr)) = (VAR, ptr, fi) and G (ptr) = (CONSTRAINT, cstr$_1$:...:cstr$_m$))
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *lb*, st:adr, rptr, binf)])
else (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st:adr, rptr, binf)])

SI ⟦JUMP_EVAL_MODE lb⟧ { jump if evaluation mode is HNF }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= if evm = HNF
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb*, st, rptr, binf)])
else (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st, rptr, binf)])

C.5 Semantics of Process Instructions

SI ⟦TRY_ME_ELSE lb⟧ { set backtracking address to lb }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st, rptr, (tr, bptr, ldpt, bad, spc))])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st, rptr, (tr, bptr, ldpt, *lb*, spc))])

SI ⟦EXECUTE⟧ { execute a function application }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:adr, rptr, (tr, bptr, ldpt, bad, spc))])
:= let adr′ := *dereference*(G,adr), v_1:...:v_k := *new*(G,k),
(TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′, DORMANT, k) := G(adr′)
in (*adr′*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *rtlb′*, st:adr, rptr, (tr, bptr, ldpt, bad, spc)),
adr′ | (TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′,
ACTIVE, *v_1:...:v_k, pc+1, ε, atp, (ε, ldpt, adr′, –, rtlb′)*),
v_1 | (UBV), ..., v_k | (UBV)])

SI ⟦EVALUATE⟧ { evaluate an application depending on evaluation mode }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:adr, rptr, binf)])
:= let adr′ := *dereference*(G,adr),
(TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′, DORMANT, k) := G(adr′)
in (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *ca′*, st:adr, rptr, binf)
adr′ | (TASK, ca′, argl′, *evm,* BDEV, cstrt′, *pc+1, atp*, DORMANT, k)])

SI ⟦ARG_EVALUATE⟧ { evaluate an application to HNF }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:adr_1:adr_2, rptr, binf)])
:= let adr' := *dereference*(G,adr_2),
(TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr', DORMANT, k) := G(adr')
in (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *ca'*, st, rptr, binf)
adr' | (TASK, ca', argl', HNF, ARGEV, cstrt', *pc+1*, adr_1, DORMANT, k)])

SI ⟦RETURN⟧ { successful return from function call }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:adr, rptr, (tr, bptr, ldpt, bad, spc)),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', pc', st', rptr', (tr', bptr', ldpt', bad', spc'))])
:= (*rptr*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
EVALUATED, locvl, *pc+1*, st:adr, rptr, (tr, bptr, ldpt, bad, spc)),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', *spc*, $\widetilde{st}$, rptr', (tr', bptr', *ldpt*, bad', spc'))])
where $\widetilde{st}$:= if cevm = ARGEV then st:fptr:atp else let $\widehat{st}$:elem := st in $\widehat{st}$:adr

SI ⟦LASTRETURN⟧ { successful return from call, no alternatives }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:adr, rptr, (tr, bptr, ldpt, bad, spc)),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', pc', st', rptr', (tr', bptr', ldpt', bad', spc'))])
bptr | (TASK, ca'', argl'', evm'', cevm'', cstrt'', rtlb'', fptr'',
ACTIVE, locvl'', pc'', st'', rptr'', (tr'', bptr'', ldpt'', bad'', spc''))])
:= if ldpt = atp {No sons}
then (*rptr*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, READY, *length(locvl), adr*)
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', *spc*, $\widetilde{st}$, rptr', (tr', bptr', *bptr*, bad', spc'))]),
bptr | (TASK, ca'', argl'', evm'', cevm'', cstrt'', rtlb'', fptr'',
ACTIVE, locvl'', pc'', st'', rptr'', (*tr'':tr*, bptr'', ldpt'', bad'', spc''))])
else {Normal Return}
(*rptr*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
EVALUATED, locvl, *pc+1*, st:adr, rptr, (tr, bptr, ldpt, bad, spc)),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', *spc*, $\widetilde{st}$, rptr', (tr', bptr', *ldpt*, bad', spc'))])
where $\widetilde{st}$ is defined like in the RETURN command.

SI ⟦FAIL_RETURN⟧ { return with failure }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st, rptr, (tr, bptr, ldpt, bad, spc)),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', pc', st', rptr', (tr', bptr', ldpt', bad', spc')),
bptr | (TASK, ca'', argl'', evm'', cstrt'', cevm'', rtlb'', fptr'',
status, locvl'', pc'', st'', rptr'', binf'')])
:= if bptr = rptr {Backtracking to the father. No brothers}
then (*rptr*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, DORMANT, *k*),
rptr | (TASK, ca', argl', evm', cevm', cstrt', rtlb', fptr',
ACTIVE, locvl', *bad'*, st', rptr', (tr', bptr', *rptr*, bad', spc'))])
else {Backtracking to a deeper node}
(*bptr*, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, DORMANT, *k*),
bptr | (TASK, ca'', argl'', evm'', cevm'', cstrt'', rtlb'', fptr'',
ACTIVE, locvl'', pc'', st'', rptr'', binf'')])

SI ⟦UNDO⟧ { undo locally trailed bindings }

(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st, rptr, (tv_1:...:tv_l, bptr, ldpt, bad, spc))])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, ε, rptr, (ε, bptr, *atp*, bad, spc)),
p_1 | $node_1$, ..., p_l | $node_l$])

where $p_i := \begin{cases} tv_i, & \underline{\text{if}}\ tv_i \in \text{Graph\_Address} \\ ptr_i, & \underline{\text{if}}\ tv_i = (ptr_i, ptr'_i, \text{fi}) \end{cases}$ and

$$\text{node}_i := \begin{cases} (\textsc{ubv}), & \underline{\text{if}}\ \text{tv}_i \in \text{Graph\_Address} \\ (\textsc{var}, \text{ptr}'_i, \text{fi}), & \underline{\text{if}}\ \text{tv}_i = (\text{ptr}_i, \text{ptr}'_i, \text{fi}) \end{cases}$$

SI ⟦FORCE⟧ { force backtracking }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st, rptr, (tr, bptr, ldpt, bad, spc)),
ldpt | (TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′, stat, locvl′, pc′, st′, rptr′, binf′)])
:= if atp = ldpt { task without sons }
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *bad*, st, rptr, (tr, bptr, ldpt, bad, spc))])
else (*ldpt*, G [ldpt | (TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′,
ACTIVE, locvl′, pc′, st′, rptr′, binf′)])

C.6 Semantics of Constraint Instructions

SI ⟦INCLUDE_CONSTRAINT⟧ { move constraint to top of stack + evaluate it}
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= if G(cstrt) = (TASK, ca′, argl′, evm′, cevm′, cstrt′, rtlb′, fptr′, DORMANT, k) then
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr, ACTIVE, locvl, *ca′*, *st:cstrt*, rptr, binf),
cstrt | (TASK, ca′, argl′, HNF, BDEV, cstrt′, *pc+1*, *atp*, DORMANT, k)])
else { cstrt = – , trivial constraint }
(atp, G[atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1*, *st:new(G,1)*, rptr, binf),
new(G,1) | (CONSTR, *true*, ε)])

A sequence of UNIFY instructions together with the INCLUDE_CONSTRAINT instruction roughly corresponds to the propagation function (see Section 4).

SI ⟦MORE⟧, SI ⟦PRINT_RESULT⟧, SI ⟦PRINT_FAILURE⟧ and SI⟦STOP⟧ are highly implementation dependent and omitted here. The remaining instructions will be specified later when they are used.

Final Configuration: (Pr, atp_f, G) with

G (atp_f) = (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st:*adr*, rptr, binf)

and Pr (pc) = STOP.

adr will point to the final result and the answer can be collected by investigating the trails. A variable X, represented by a node with address p, has to be bound to a *finite* value, if it exists a variable represented by a node with address p' and *dereference*(G,p) = *dereference*(G,p') and (G(p') = (VAR, ptr, FIN) or if G(p) = (CONSTR, c, *ptrs*) and all the components (represented by *ptrs*) have to be finite. Otherwise, X may be bound to an infinite value.

D Predefined Operations

First, we define the binary function "eq", which either produces *true*, if both arguments are finite and equal, or fails.

```
ca(eq): EXECUTE
        TRY_ME_ELSE fl
        LOAD 2
        LOAD 1
l0:     JUMP_HNF l1
        EVALUATE
l1:     SWAP
        JUMP_HNF l2
        EVALUATE
l2:     CHECKEQ
        JPES l3
        JUMPl0
l3:     INCLUDE_CONSTRAINT
        RETURN
fl:     UNDO
        FAIL_RETURN
```

```
ca(neq): EXECUTE
         TRY_ME_ELSE fl
         LOAD 1
l0:      JUMP_HNF l1
         EVALUATE
         LOAD 2
         JUMP_HNF l2
         EVALUATE
l2:      CHECKDISEQ
         INCLUDE_CONSTRAINT
         RETURN
fl:      UNDO
         FAIL_RETURN
```

The new commands CHECKEQ, CHECKDISEQ, JPES, and SWAP are formally specified afterwards.

"eq" and "neq" can be used to define the equality "=" and disequality "≠" by the following Babel rules:

fun =: $\alpha \times \alpha \rightarrow$ *bool*.
  (X = Y) := (X eq Y) → *true*.
  (X = Y) := (X neq Y) → *false*.

fun ≠: $\alpha \times \alpha \rightarrow$ *bool*.
  (X ≠ Y) := (X neq Y) → *true*.
  (X ≠ Y) := (X eq Y) → *false*.

Moreover, the functions bool_id, true_gen, false_gen, and nfe (normal form evaluation) are predefined by the following Babel rules. The rules for the operations ¬, _→_ (if-then), and _→_□_ (if-then-else) have already been shown in Subsection 2.4.

fun bool_id: *bool* → *bool*.
  bool_id *true* := *true*.
  bool_id *false* := *false*.

fun true_gen: *bool* → *bool*.
  true_gen *true* := *true*.

fun false_gen: *bool* → *bool*.
  false_gen *false* := *false*.

fun nfe: $\alpha \rightarrow \alpha$.
  nfe X := (X = X) → X.

D.1 Commands CHECKEQ, CHECKDISEQ, JPES, and SWAP

The semantics of the commands CHECKEQ and CHECKDISEQ corresponds to the rules for equality and disequality in Section 4.

SI ⟦CHECKEQ⟧ { check 2 top elements on stack for equality }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
  ACTIVE, locvl, pc, st:$\widehat{v_1}$: $\widehat{v_2}$, rptr, (tr, bptr, ldpt, bad, spc))]) :=
let v_1 := *dereference*(G,$\widehat{v_1}$), v_2 := *dereference*(G,$\widehat{v_2}$) in
if G(v_1) = (UBV) then
  if $v_1 = v_2$ then { see rule (R4.5) in Section 4 }
    (atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
      ACTIVE, locvl, *pc+1*, st, rptr, (tr, bptr, ldpt, bad, spc))])
  else if G(v_2) = (CONSTR, c, p_1:...:p_n) and not *term*(G,v_2) then { see (R4.4) }
      let p'_1:...:p'_n := *new*(G,n) in
      (atp, G [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
        ACTIVE, locvl, *pc+1*, *st*:p_1:p'_1:...:p_n:p'_n, rptr, (*tr*:v_1, bptr, ldpt, bad, spc),
        v_1 | (VAR, v_2, FIN),
        p'_1 | (UBV), ..., p'_n | (UBV)])
    else { see (R4.6) }
      (atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
        ACTIVE, locvl, *pc+1*, *st*, rptr, (*tr*:v_1, bptr, ldpt, bad, spc),
    v_1 | (VAR, v_2, FIN)])
else if G(v_2) = (UBV) then analogously to case G(v_1) = (UBV)
else if G(v_1) = (CONSTR, c, p_1:...:p_n) then
    if G(v_2) = (CONSTR, c′, p'_1:...:p'_m) then
      if c = c′ and n = m then { see (R4.3) }
        (atp, G [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
          ACTIVE, locvl, *pc+1*, *st*:p_1:p'_1:...:p_n:p'_n, rptr, (tr, bptr, ldpt, bad, spc))])
      else { see (F4.1) }
        (atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
          ACTIVE, locvl, *bad*, ε, rptr, (tr, bptr, ldpt, bad, spc))])
    else let (VAR, ptr, fi) := G(v_2), (CONSTRAINT, $cstr_1$:...:$cstr_m$) := G(ptr) in
        if not *term*(G,v_1) then { see (R4.4) }
          let p'_1:...:p'_n := *new*(G,n) in
          (atp, G [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
            ACTIVE, locvl, *pc+1*, *st*:p_1:p'_1:...:p_n:p'_n, rptr, (*tr*:(v_2,*ptr*,*fi*), bptr, ldpt, bad, spc),
            v_2 | (VAR, v_1, FIN),
            p'_1 | (UBV), ..., p'_n | (UBV)])
        else { see (R4.6) }
          (atp, G' [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
            ACTIVE, locvl, *pc+1*, *st*, rptr, (*tr*:(v_2, *ptr*, *fi*), bptr, ldpt, bad, spc),
            v_2 | (VAR, v_1, FIN)])
        where (cstrt, G′) := *build*(G,c,$cstr_1$:...:$cstr_m$, v_1)
else let (VAR, ptr, fi) := G(v_1), (CONSTRAINT, $cstr_1$:...:$cstr_m$) := G(ptr) in
    if G(v_2) = (CONSTR, c′, p_1:...:p_n) then
      analogously to case G(v_1) = (CONSTR, c, p_1:...:p_n), G(v_2) = (VAR, ptr, fi)

else let (VAR, ptr', fi') := G(v_2), (CONSTRAINT, $cstr'_1$:...:$cstr'_n$) := G(ptr')
in if $v_1 = v_2$ then { see (R4.5) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st, rptr, (tr, bptr, ldpt, bad, spc)])
else if ptr $\in \{$*dereference*(G,$cstr'_j$)$| \ 1 \leq j \leq n\}$ or
ptr' $\in \{$*dereference*(G,$cstr_j$)$| \ 1 \leq j \leq m\}$ then
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *bad*, ε, rptr, (tr, bptr, ldpt, bad, spc)])
else {see (R4.6) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr,
(tr:(v_1, ptr, fi):(v_2, ptr', fi'), bptr, ldpt, bad, spc),
v_1 | (VAR, v_2, FIN), v_2 | (VAR, *new*(G,1), FIN),
new(G,1) | (CONSTRAINT, $cstr_1$:...:$cstr_m$:$cstr'_1$:...:$cstr'_n$)])

To avoid a further complication, an occur check is omitted. Moreover, let us in the following ignore the problem of data types with a finite number of values. For such data types, a second version of the CHECK-DISEQ instruction (and the neq-function) is needed, which takes into account the rules (R5.9), (R5.10), and (R5.11) of Section 4. The compiler has to use the correct version depending on the type of the expressions, which shall be compared.

SI ⟦CHECKDISEQ⟧ { check 2 top elements on stack for disequality }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, pc, st:$\widehat{v_1}$: $\widehat{v_2}$, rptr, (tr, bptr, ldpt, bad, spc)]) :=
let v_1 := *dereference*(G,$\widehat{v_1}$), v_2 := *dereference*(G,$\widehat{v_2}$) in
if G(v_1) = (UBV) then
if G(v_2) = (CONSTR, c, p_1:...:p_{n_0}) then
if *term*(G,v_2) then { see (R5.8) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (*tr:v_1*, bptr, ldpt, bad, spc)),
v_1 | (VAR, *new*(G,1), INF), *new*(G,1) | (CONSTRAINT, v_2)])
else { see rules (R5.5) and (R5.6) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (tr, bptr, ldpt, bad, spc),
cstrt | (TASK, ca(or), new_0:new_1:...:new_k, –, –, –, –, –, DORMANT,0),
new_0 | (TASK, ca(and), $\widehat{new}$:ornode, –, –, –, –, –, DORMANT,0),
new_1 | (TASK, ca(eq), v_1:new'_1, –, –, –, –, –, DORMANT,0),
⋮
new_k | (TASK, ca(eq), v_1:new'_k, –, –, –, –, –, DORMANT,0),
$\widehat{new}$ | (TASK, ca(eq), v_1:new'_0, –, –, –, –, –, DORMANT,0),
new'_0 | (CONSTR, c, var_1:...:var_{n_0}),
new'_1 | (CONSTR, c_1, var_1:...:var_{n_1}),
⋮
new'_k | (CONSTR, c_k, var_1:...:var_{n_k}),
ornode | (TASK, ca(or), $comp_1$:...:$comp_{n_0}$, –, –, –, –, –, DORMANT,0),
$comp_1$ | (TASK, ca(neq), var_1:p_1, –, –, –, –, –, DORMANT,0),
⋮
$comp_{n_0}$ | (TASK, ca(neq), var_{n_0}:p_{n_0}, –, –, –, –, –, DORMANT,0)])
var_1 | (UBV), ..., var_{mx} | (UBV)])
where $c_1,\ldots,c_k$ are the constructors (except c) with the same result type as c,
mx := $\max\{n_0, \ldots, n_k\}$,
cstrt:new_0:...:new_k:$\widehat{new}$:new'_0:...:new'_k:ornode:$comp_1$:...:$comp_{n_0}$:var_1:...:var_{mx} :=
new(G, $2 * k + n_0 + mx + 5$)
else if G(v_2) = (UBV) then
if $v_1 = v_2$ then
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *bad*, ε, rptr, (tr, bptr, ldpt, bad, spc)])
else let v'_1:v'_2 := *new*(G,2) in { see rule (R5.7) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (*tr:v_2:v_1*, bptr, ldpt, bad, spc),
v_1 | (VAR, v'_1, INF), v_2 | (VAR, v'_2, INF),
v'_1 | (CONSTRAINT, v_2), v'_2 | (CONSTRAINT, v_1)])
else if G(v_2) = (VAR, ptr, fi) and G(ptr) = (CONSTRAINT, t_1:...:t_n) then { see rule (R5.7) }
let v'_1:v'_2 := *new*(G,2) in

(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (*tr:*(v_2,ptr,fi)*:*v_1, bptr, ldpt, bad, spc),
v_1 | (VAR, v'_1, INF), v_2 | (VAR, v'_2, fi),
v'_1 | (CONSTRAINT, v_2), v'_2 | (CONSTRAINT, t_1:...:t_n:v_1)])
else if G(v_2) = (UBV) then analogously to case G(v_1) = (UBV)
else if G(v_1) = (CONSTR, c, p_1:...:p_n) then
if G(v_2) = (CONSTR, c′, p'_1:...:p'_m) then
if c = c′ and n = m then { see (R5.4) }
let cstrt:new_1:...:new_n := *new*(G,n+1) in
(atp, G [atp | (TASK, ca, argl, evm, cevm, *cstrt*, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (tr, bptr, ldpt, bad, spc),
cstrt | (TASK, ca(or), new_1:...:new_n, –, –, –, –, –, DORMANT,0),
new_1 | (TASK, ca(neq), p_1:p'_1, –, –, –, –, –, DORMANT,0),
⋮
new_n | (TASK, ca(neq), p_n:p'_n, –, –, –, –, –, DORMANT,0)])
else { see rule (R5.3) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (tr, bptr, ldpt, bad, spc)])
else let (VAR, ptr, fi) := G(v_2), (CONSTRAINT, ptrs) := G(ptr) in
analogously to case G(v_2) = (CONSTR, c′, p_1:...:p_n), G(v_1) = (VAR, ptr, fi), see below
else let (VAR, ptr, fi) := G(v_1), (CONSTRAINT, ptrs) := G(ptr) in
if G(v_2) = (CONSTR, c′, p_1:...:p_n) then
if *term*(G,v_2) then { see (R5.8) }
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, (*tr:*(v_1,ptr,fi), bptr, ldpt, bad, spc)),
v_1 | (VAR, *new*(G,1), INF), *new*(G,1) | (CONSTRAINT, v_2:ptrs)])
else like case G(v_1) = (UBV), G(v_2) = (CONSTR, c, p_1:...:p_{n_0})
else let (VAR, ptr′, fi′) := G(v_2), (CONSTRAINT, ptrs′) := G(ptr′) in
if v_1 = v_2 then
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *bad*, ε, rptr, (tr, bptr, ldpt, bad, spc)])
else { see (R5.7) }
let v'_1:v'_2 := *new*(G,2) in
(atp, G [atp | (TASK, ca, argl, evm, cevm, –, rtlb, fptr,
ACTIVE, locvl, *pc+1, st*, rptr, *(tr:*(v_1, ptr, fi)*:*(v_2, ptr', fi')*)*, bptr, ldpt, bad, spc),
v_1 | (VAR, v'_1, fi), v_2 | (VAR, v_2, fi′),
v'_1 | (CONSTRAINT, ptrs:v_2), v'_2 | (CONSTRAINT, ptrs′:v_1)])

SI ⟦JPES lb⟧ { jump if empty stack }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= if st = ε
then (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, *lb*, st, rptr, binf)])
else (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, st, rptr, binf)])

SI ⟦SWAP⟧ { exchange two elements on top of stack }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, pc, st:adr_1:adr_2, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1*, *st:*adr_2*:*adr_1, rptr, binf)])

D.2 Predefined Operations used with Constraints

During the unification of a function application with the left hand side of a rule, a constraint is built (see UNIFY instruction). This constraint restricts the applicability of the rule. It is represented as a boolean expression. This expression must be evaluated to *true*. If this is not possible, it should fail. Hence, we cannot use the ordinary conjunction $\wedge$ and disjunction $\vee$ but a special n-ary conjunction "and" and disjunction "or" respectively which either delivers *true* or fails. The corresponding code is shown below. Both pieces of code use the new command LOADARGS, which loads the arguments to the stack.

ca(and): EXECUTE
TRY_ME_ELSE fl
LOADARGS
l_1: JPES l'
JUMP_HNF l''
EVALUATE
l'': JUMP_FALSE force_lb
JUMP l_1
l': CONSTRNODE (*true*,0)
LASTRETURN
fl: FAIL_RETURN

ca(or): EXECUTE
TRY_ME_ELSE fl
LOADARGS
l_1: JPES fl
JUMP_HNF l''
EVALUATE
l'': JUMP_FALSE force_lb
CONSTRNODE (*true*,0)
RETURN
fl: FAIL_RETURN

Note that "or" is lazy in the sense that it evaluates its arguments from left to right until one of them is evaluated to *true*. The remaining arguments are not (yet) evaluated. The LOADARGS instruction is defined as follows:

SI [[LOADARGS]] { load arguments to stack }
(atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr, ACTIVE, locvl, pc, st, rptr, binf)])
:= (atp, G [atp | (TASK, ca, argl, evm, cevm, cstrt, rtlb, fptr,
ACTIVE, locvl, *pc+1, st:argl*, rptr, binf)])

E Result Directed Computing

There are several possibilities to achieve some kind of result directed computing avoiding the production of useless results. One possibility is to build it into the machine. This approach is the most flexible, but causes some runtime overhead. Another possibility is to use a program transformation [KL92]. This approach was sketched in Section 4.4. Alternatively, a more sophisticated code generation scheme can be applied. In principle, the code for the transformed program can be directly produced by such a scheme without needing an explicit transformation. Here, we will only sketch this idea for boolean expressions. The presented scheme can surely be improved, e.g. by propagating the fail label fl in the *exptrans* scheme.

The translation of guarded expressions can be changed to:

exptrans ($b \rightarrow e$) := *ctrans*(b, *true*, force_lb)
exptrans(e)

The new scheme *ctrans* is defined as follows:

ctrans: Exp $\times$ Bool $\times$ Program_Address $\rightarrow$ Code

The second argument is a tag representing the desired result. If this result cannot be produced, a jump to the label given as the 3rd argument will be performed. In contrast to *exptrans*, *ctrans* does *not* push the result onto the stack.

$ctrans(\neg b,\, bv,\, \mathrm{fl}) := ctrans(b,\, \overline{bv},\, \mathrm{fl})$

$ctrans(b_1 \wedge b_2,\, true,\, \mathrm{fl}) :=$
$\quad ctrans(b_1,\, true,\, \mathrm{fl})$
$\quad ctrans(b_2,\, true,\, \mathrm{fl})$

$ctrans(b_1 \wedge b_2,\, false,\, \mathrm{fl}) :=$
$\quad ctrans(b_1,\, false,\, \mathrm{l})$
JUMP end
l: $ctrans(b_2,\, false,\, \mathrm{fl})$
end: ...

$ctrans(b_1 \vee b_2,\, false,\, \mathrm{fl}) :=$
$\quad ctrans(b_1,\, false,\, \mathrm{fl})$
$\quad ctrans(b_2,\, false,\, \mathrm{fl})$

$ctrans(b_1 \vee b_2,\, true,\, \mathrm{fl}) :=$
˜ $=ctrans(b_1,\, true,\, \mathrm{l})$
JUMP end
l:$ctrans(b_2,\, true,\, \mathrm{fl})$
end:

$ctrans\ (e_1 = e_2,\, true,\, \mathrm{fl}) :=$
TRY_ME_ELSE fl
$graphtrans(e_1)$
$graphtrans(e_2)$
NODE (ca(eq),2,0)
EVALUATE
POP

$ctrans\ (e_1 = e_2,\, false,\, \mathrm{fl}) :=$
TRY_ME_ELSE fl
$graphtrans(e_1)$
$graphtrans(e_2)$
NODE (ca(neq),2,0)
EVALUATE
POP

$ctrans\ (e_1 \neq e_2,\, bv,\, \mathrm{fl}) :=$
$\quad ctrans\ (e_1 = e_2,\, \overline{bv},\, \mathrm{fl})$

$ctrans\ (e,\, true,\, \mathrm{fl}) :=$
TRY_ME_ELSE fl
$exptrans(e)$
JPNONVAR l_1
NODE (ca(true_gen), 1, 0)
EVALUATE
l_1: JUMP_FALSE force_lb
if no previous definition applies

$ctrans\ (e,\, false,\, \mathrm{fl}) :=$
TRY_ME_ELSE fl
$exptrans(e)$
JPNONVAR l_1
NODE (ca(false_gen), 1, 0)
EVALUATE
l_1: JUMP_TRUE force_lb
if no previous definition applies