ISLIP 98

Proceedings

Eleventh Annual International Symposium on Languages for Intensional Programming

Bill Wadge, Editor

Sun Microsystems

Palo Alto, California, USA

May 7-9, 1998

Transformation Techniques for Branching-time Logic Programs*

Manolis Gergatsoulis, Costas Spyropoulos

Inst. of Informatics & Telecom., N.C.S.R. 'Demokritos',
153 10 A. Paraskevi Attikis, Greece
e_mail: {manolis,costass}@iit.nrcps.ariadne-t.gr

Abstract

In this paper, we consider program transformation techniques for branching-time logic programs. We define a set of unfold/fold transformation rules and present sufficient conditions to ensure their correctness. Then, using the proposed transformation rules we develop an algorithm which transforms a wide class of Cactus programs into a continuation passing style form.

Keywords: Program Transformation, Logic Programming, Temporal Logic Programming, Branching Time, Continuation Passing Style.

1 Introduction

A lot of research effort has been devoted recently in developing logic programming languages which incorporate, in one or another way, the notion of time [Org91, OM94, Bau93, RGP97b, GRP97]. Most of the temporal logic programming languages presented in the literature are based on linear flow of time [OM94, Hry93, OWD93, Bau93, Brz91, Brz93, GRP96]. However, in [RGP97b, RGP97a, GRP97] a temporal logic programming language called **Cactus**, which is based on a tree-like notion of time, was introduced. In Cactus, there is an initial moment in time and every moment may have more than one next moments.

Temporal logic programming languages are recognized as natural and expressive formalisms for describing *dynamic* systems. In particular, branching-time logic programming is useful for describing non-deterministic computations as well as computations that involve the manipulation of trees.

On the other hand, program transformation techniques have been widely used in program synthesis [ST84, PP94b], program optimization [PP95, Ger94, AGK96], program specialization [BCD90] and partial evaluation (partial deduction) [LS91, PP93]. Program transformation systems, based on unfold/fold rules, have been developed for functional programs [BD77], as well as for definite and normal (programs permitting negative subgoals) logic programs [TS84, TS86, KK90, BC93, PP94a, GK94, Sek91, Sek93]. For good surveys on program transformation of definite and normal logic programs one may refer in [PP97, PP94a].

*This work has been partially supported by the Greek General Secretariat of Research and Technology under the project "Logic Programming Systems and Environments for developing Logic Programs" of ΠΕΝΕΔ'95, contract no 952.

In this paper we define a program transformation system for branching-time (Cactus) logic programs and present sufficient conditions for its correctness. The main rules of the system namely *unfolding* and *folding* are extensions of the corresponding rules [TS86, Ger94] for definite clause programs. A transformation rule, called *temporal shift* which takes into account the properties of time, is also presented.

As an application of the proposed transformation system, we develop an algorithm which compiles a wide class of branching time logic programs into a continuation passing style form.

The rest of the paper is organized as follows: in section 2, we briefly present the branching-time logic programming language Cactus. In section 3, we introduce the transformation rules, and in section 4 we propose sufficient conditions for their correctness. In section 5, based on the transformation system we develop a continuation passing style transformation algorithm for Cactus programs. Finally, section 6 concludes the paper.

2 The *Cactus* branching-time logic programming language

The syntax of Cactus programs extends the syntax of Definite Clause programs [Llo87]. A *Cactus program* is a finite set of *temporal clauses*. A *temporal clause* is a formula of the form:

$$H \leftarrow B_1, \ldots, B_m$$

where $m \geq 0$ and $H, B_1, \ldots, B_m$ are *temporal atoms*. A temporal atom is a classical atom preceded by a (possibly empty) sequence of temporal operators. The sequence of operators of a temporal atom is called the *temporal reference* of that atom. If $m = 0$ then the clause is said to be a *unit temporal clause*.

A *goal clause* in Cactus is a formula of the form $\leftarrow A_1, \ldots, A_n$ where A_i, $i = 1, ..., n$ are temporal atoms.

Cactus supports two temporal operators: the temporal operator `first` which refers to the beginning of time (or alternatively to the root of the time-tree), and the temporal operator $\texttt{next}_i$ which refers to the i-th child of the current moment (or alternatively, the i-th branch of the current node in the tree). Notice that we actually have a family $\{\texttt{next}_i \mid i \in N\}$ of `next` operators, each one of them representing a different next moment that immediately follows the present one.

Example 2.1. Consider the following Cactus program

$$\begin{array}{lll} \texttt{p(0)}. & & \\ \texttt{next}_0\ \texttt{p(s(X))} & \leftarrow & \texttt{p(X)}. \\ \texttt{next}_1\ \texttt{p(s(X))} & \leftarrow & \texttt{next}_0\ \texttt{p(X)}. \end{array}$$

which defines the non-deterministic predicate '`p`'. The sets of values of the argument of '`p`' for which the predicate is true at each moment in the time tree are shown in figure 1.

For example the query

$$\leftarrow\ \texttt{first}\ \texttt{next}_0\ \texttt{next}_1\ \texttt{p(X)}.$$

will return the following answers (values of the variable '`X`'):

$$\begin{array}{l} \texttt{X} = \texttt{0} \\ \texttt{X} = \texttt{s(0)} \\ \texttt{X} = \texttt{s(s(0))} \\ \texttt{X} = \texttt{s(s(s(0)))} \end{array}$$

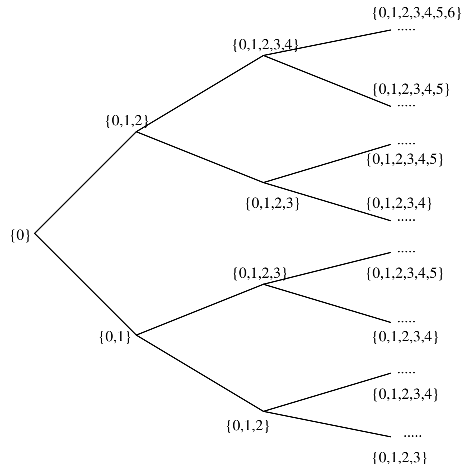

Figure 1: Part of the time-tree and the corresponding sets of values (in Arabic style numbering) of the argument of '`p`' for which the predicate '`p`' is true.

A temporal reference T is said to be *canonical* if the leftmost operator of T is `first`.

The semantics of Cactus programs are defined in [RGP97b, GRP97]. It is important to note here that a Cactus program P has a *least Herbrand model* $M(P)$ which comprises all canonical temporal atoms which are logical consequences of P.

In the following sections we will also use the notion of *temporal unifiers*. Temporal unifiers have been introduced in [GRP97], and extend the notion of unifier in classical logic. Temporal unification requires that temporal references are in *normal form*. We say that a temporal reference T is in normal form, if either the operator `first` does not appear in T or there is only one occurrence of `first` in T, which is the leftmost operator in T. Every temporal reference T can be transformed into normal form $normal(T)$ by eliminating all operators appearing before the rightmost occurrence of `first`. The axioms of branching time logic of Cactus [RGP97b] ensure that for every formula A we have: $T\ A \leftrightarrow normal(T)\ A$. Intuitively, normalization consists in discarding redundant temporal operators. For example, the normal form of the temporal atom $\texttt{next}_2\ \texttt{first}\ \texttt{next}_0\ \texttt{first}\ \texttt{next}_1\ \texttt{A}$ is $\texttt{first}\ \texttt{next}_1\ \texttt{A}$.

Finally, by $T_1\ T_2$ we denote the temporal reference obtained by putting the temporal reference T_1 before the temporal reference T_2. We say that $T_1\ T_2$ is the *composition* of the temporal references T_1 and T_2.

Definition 2.1 (*Temporal Unifier*). Let A_1 and A_2 be two temporal atoms, such that $A_1 = R_1\ A'_1$ and $A_2 = R_2\ A'_2$ where A'_1 and A'_2 are classical atoms and R_1, R_2 are (possibly empty[1]) temporal references in normal form. Then $\theta^t = (T, \theta, S)$, where T and S are temporal references in normal form and θ is a substitution, is said to be a *temporal unifier* of A_1 and A_2 iff $T\ R_1\ A'_1\theta = S\ R_2\ A'_2\theta$ and both $T\ R_1$ and $S\ R_2$ are also in normal form. Two temporal atoms A_1 and A_2 are said to be *temporally unifiable* iff they have a temporal unifier.

Definition 2.2 (*Most general Temporal Unifier*). A temporal unifier $\theta^t = (T, \theta, S)$ of two temporal atoms A_1 and A_2, is said to be a *most general temporal unifier* of A_1 and A_2 (we write $\theta^t = mgu^t(A_1, A_2)$) iff for every unifier $\sigma^t = (T', \sigma, S')$ of A_1 and A_2, there is a temporal substitution $\xi^t = (T'', \xi, S'')$ such that $\sigma = \theta\xi$, $T' = T''\ T$, and $S' = S''\ S$.

[1]We denote an empty temporal reference by ϵ.

It is easy to see that $(T, \theta, S) = mgu^t(A_1, A_2)$ iff $(S, \theta, T) = mgu^t(A_2, A_1)$. Moreover, if two temporal atoms are temporally unifiable, they have a most general temporal unifier.

3 The transformation Rules

In general, the transformation process starts from a program P_0, called the *initial program*, and produces a sequence of programs $P_0, P_1, \ldots, P_i, \ldots$, called a *transformation sequence*, such that each program in the sequence is obtained by applying a transformation rule to the preceding one.

Definition 3.1 (*Initial Program*). An *initial program* P_0, is a Cactus program satisfying the following conditions:

1. Let $\mathcal{P}$ be the set of predicates of P_0. Then $\mathcal{P}$ is divided into two disjoint sets $\mathcal{P}^p$ and $\mathcal{P}^t$. The predicates in $\mathcal{P}^p$ are called *primitive predicates* while the predicates in $\mathcal{P}^t$ are called *transformable predicates.*
2. Transformable predicates do not appear in the bodies of the clauses defining primitive predicates.

Definition 3.1 implies that an initial program P_0 is divided into two disjoint sets of clauses, P_0^p and P_0^t. P_0^p comprises the definitions of the primitive predicates while P_0^t comprises the definitions of the transformable predicates. Will see in the following that the transformation rules are applied only to clauses defining transformable predicates. As a consequence, each program P_l, with $l \geq 0$, in the transformation sequence consists also of two distinct sets of clauses P_l^p and P_l^t. Since no transformation rule is applied to the clauses defining primitive predicates, P_l^p is identical to P_0^p.

Although in practice, new predicate definitions (*Eureka* definitions) are often introduced during the transformation process, these definitions are considered as part of the initial program.

Example 3.1. Let $P_0 = \{1, 2, 3, 4, 5, 6\}$ be the following Cactus program:

(1)	$\texttt{even\_p(X)} \leftarrow \texttt{p(X), even(X)}.$
(2)	$\texttt{p(0)}.$
(3)	$\texttt{next}_0\ \texttt{p(s(X))} \leftarrow \texttt{p(X)}.$
(4)	$\texttt{next}_1\ \texttt{p(s(X))} \leftarrow \texttt{next}_0\ \texttt{p(X)}.$
(5)	$\texttt{even(0)}.$
(6)	$\texttt{even(s(s(X)))} \leftarrow \texttt{even(X)}.$

P_0 can be seen as an initial program with $P_0^t = \{1, 2, 3, 4\}$ and $P_0^p = \{5, 6\}$. Notice that the clauses $\{2, 3, 4\}$ define the predicate '`p`' as in example 2.1. Clauses $\{5, 6\}$ define the predicate '`even`' which is true for the even numbers. Finally, '`even_p`' is true at a moment in time, only for the even values of the argument for which '`p`' is true at the same moment.

Definition 3.2 (*Unfolding*). Let C be a clause in P_l^t of the form

$C: \qquad A \leftarrow M, B, N$

where A, B are temporal atoms and M, N are (possibly empty) conjunctions of temporal atoms. Let $D_1, \ldots, D_m$ be all clauses in a program P_k, with $0 \leq k \leq l$, whose heads are temporally unifiable with B by the most general temporal unifiers $\theta_1^t, \ldots, \theta_m^t$ respectively. If (at least) one of the following conditions holds:

1. A, M, and N are canonical.
2. B is not canonical.
3. For each clause D_j, with $1 \leq j \leq m$, either all body atoms of D_j are canonical, or the head of D_j is not canonical.

Then, the result of *unfolding C at B* consists in replacing C in P_l by the set of clauses $\{C'_1, \ldots, C'_m\}$ constructed as follows: for each j, with $1 \leq j \leq m$, if

$D_j : \qquad B_j \leftarrow G$

where G is a (possibly empty) conjunction of temporal atoms, then

$C'_j : \qquad (T_j\ A \leftarrow T_j\ M, S_j\ G, T_j\ N)\theta_j$

where $\theta_j^t = (T_j, \theta_j, S_j) = mgu^t(B, B_j)$. C is called the *unfolded clause* and $C_1, \ldots, C_m$ are called the *unfolding clauses*.

Example 3.2 (*Continued from example 3.1*). Unfolding clause (1) at '`p(X)`' using clauses $\{2, 3, 4\}$ we get:

(7) $\quad$ `even_p(0)` $\leftarrow$ `even(0).`
(8) $\quad$ $\texttt{next}_0$ `even_p(s(X))` $\leftarrow$ `p(X),`$\texttt{next}_0$ `even(s(X)).`
(9) $\quad$ $\texttt{next}_1$ `even_p(s(X))` $\leftarrow$ $\texttt{next}_0$ `p(X),`$\texttt{next}_1$ `even(s(X)).`

Unfolding clause (7) using (5) we get:

(10) $\quad$ `even_p(0).`

Unfolding clause (8) at '$\texttt{next}_0$ `even(s(X))`' using (6), and clause (9) at '$\texttt{next}_1$ `even(s(X))`' using (6) we get:

(11) $\quad$ $\texttt{next}_0$ `even_p(s(s(X)))` $\leftarrow$ `p(s(X)),`$\texttt{next}_0$ `even(X).`
(12) $\quad$ $\texttt{next}_1$ `even_p(s(s(X)))` $\leftarrow$ $\texttt{next}_0$ `p(s(X)),`$\texttt{next}_1$ `even(X).`

Unfolding clause (11) at '`p(s(X))`' using $\{3, 4\}$ we get:

(13) $\quad$ $\texttt{next}_0$ $\texttt{next}_0$ `even_p(s(s(X)))` $\leftarrow$ `p(X),`$\texttt{next}_0$ $\texttt{next}_0$ `even(X).`
(14) $\quad$ $\texttt{next}_1$ $\texttt{next}_0$ `even_p(s(s(X)))` $\leftarrow$ $\texttt{next}_0$ `p(X),`$\texttt{next}_1$ $\texttt{next}_0$ `even(X).`

Unfolding clause (12) at '$\texttt{next}_0$ `p(s(X))`' using (3) we get:

(15) $\quad$ $\texttt{next}_1$ `even_p(s(s(X)))` $\leftarrow$ `p(X),`$\texttt{next}_1$ `even(X).`

The violation of the conditions 1-3 in definition 3.2, destroys the equivalence of programs, as shown in the following example:

Example 3.3. Let P be the following program:

(1) $\quad$ `first p` $\leftarrow$ `first q,r.`
(2) $\quad$ `first q` $\leftarrow$ `s.`
(3) $\quad$ `first r.`
(4) $\quad$ `first` $\texttt{next}_0$ `s.`

Unfolding (1) at `first q` using clause (2), we get the program $P_1 = \{2, 3, 4, 5\}$, where:

(5) $\quad \texttt{first p} \ \leftarrow \ \texttt{s,r}.$

It is easy to see that $M(P) = \{\texttt{first r}, \texttt{first next}_0\ \texttt{s}, \texttt{first q}, \texttt{first p}\}$, while $M(P_1) = \{\texttt{first r}, \texttt{first next}_0\ \texttt{s}, \texttt{first q}\}$. Hence $M(P) \neq M(P_1)$.

Definition 3.3 (*Rigid predicate*). A predicate p is said to be rigid if it does not depends on time i.e. every ground instance of p is either true in all moments in time or false in all moments in time. A predicate p is said to be *syntactically rigid* (or *s-rigid* for short) if all predicates on which p depends on, are defined by operator-free clauses[2].

Definition 3.4 (*Temporal Shift*). Let C be a clause in P_l^t of the form
$C: \quad A \leftarrow M, B, N$
where A, B are temporal atoms, and M, N are (possibly empty) conjunctions of temporal atoms. Let B be of the form $Tref\ B'$, where the temporal reference $Tref$ may be empty. Let C' be a clause obtained by replacing $Tref\ B'$ in the body of C by $Tref'B'$. We say that C' is obtained by applying the *temporal shift* transformation rule to C, if

1. The predicate of B' is rigid, and
2. Either
   (a) B' is primitive, or
   (b) B' is also s-rigid in P_0.

Example 3.4 (*Continued from example 3.2*). Since the predicate '`even`' is s-rigid we can eliminate the temporal reference '$\texttt{next}_0\ \texttt{next}_0$' of the atom '$\texttt{next}_0\ \texttt{next}_0\ \texttt{even(X)}$' by applying the temporal shift transformation rule in the body of (13). We obtain

(16) $\quad \texttt{next}_0\ \texttt{next}_0\ \texttt{even\_p(s(s(X)))} \ \leftarrow \ \texttt{p(X)}, \texttt{even(X)}.$

In the same way, we can replace the temporal reference '$\texttt{next}_1\ \texttt{next}_0$' of the temporal atom '$\texttt{next}_1\ \texttt{next}_0\ \texttt{even(X)}$' in the body of (14), by the temporal reference '$\texttt{next}_0$'. We get

(17) $\quad \texttt{next}_1\ \texttt{next}_0\ \texttt{even\_p(s(s(X)))} \ \leftarrow \ \texttt{next}_0\ \texttt{p(X)}, \texttt{next}_0\ \texttt{even(X)}.$

Finally, by applying the temporal shift rule to the clause (15) we get

(18) $\quad \texttt{next}_1\ \texttt{even\_p(s(s(X)))} \ \leftarrow \ \texttt{p(X)}, \texttt{even(X)}.$

Definition 3.5 (*Folding*). Let C be a clause in P_l^t of the form
$C: \quad A \leftarrow M, F, N$
and D be a clause in P_0^t of the form:
$D: \quad B \leftarrow G$
where A and B are temporal atoms and M, F, N, G are (possibly empty) conjunctions of atoms. Then *folding C using D* consists in replacing C in P_l by the clause C', where:
$C': \quad A \leftarrow M, T\ B\theta, N$
iff the following conditions hold:

[2] I.e. clauses with no temporal operator applied to their atoms.

1. There exists a temporal reference T and a substitution θ such that:
   (a) $T\ G\theta = F$, and
   (b) θ maps the variables which occur in the body of D but not in the head of D, into distinct variables which do not occur in C'.
2. (At least) one of the following holds:
   (a) A, M and N are canonical.
   (b) G is canonical.
   (c) B is not canonical.
3. D is the only clause in P_0 whose head is temporally unifiable with $T\ B\theta$.

C is called a *folded clause*, D is called the *folding clause* and $T\ B_0\theta$ the *atom introduced by folding*.

Example 3.5 (*Continued from example 3.4*). Folding (16) using (1) we get

(19) $\quad \texttt{next}_0\ \texttt{next}_0\ \texttt{even\_p(s(s(X)))} \leftarrow \texttt{even\_p(X)}.$

Folding (17) using (1) we get

(20) $\quad \texttt{next}_1\ \texttt{next}_0\ \texttt{even\_p(s(s(X)))} \leftarrow \texttt{next}_0\ \texttt{even\_p(X)}.$

Finally, folding (18) using (1) we obtain

(21) $\quad \texttt{next}_1\ \texttt{even\_p(s(s(X)))} \leftarrow \texttt{even\_p(X)}.$

Violation of condition 2 in definition 3.5, destroys the equivalence of programs:

Example 3.6. Let P be the following program:

(1) $\quad \texttt{p} \leftarrow \texttt{q}, \texttt{r}.$
(2) $\quad \texttt{first s} \leftarrow \texttt{r}.$
(3) $\quad \texttt{first q}.$
(4) $\quad \texttt{first next}_0\ \texttt{r}.$

Then $M(P) = \{\texttt{first q}, \texttt{first next}_0\ \texttt{r}, \texttt{first s}\}$. Let us fold (violating condition 2) clause (1) using clause (2). We get the program $P_1 = \{2, 3, 4, 5\}$, where:

(5) $\quad \texttt{p} \leftarrow \texttt{q}, \texttt{first s}.$

Then $M(P_1) = \{\texttt{first q}, \texttt{first next}_0\ \texttt{r}, \texttt{first s}, \texttt{first p}\}$. Thus $M(P_1) \neq M(P)$.

Definition 3.6 (*Transformation Sequence*). A sequence of programs $P_0, P_1, \ldots, P_l$ is called a *transformation sequence starting from the initial program* P_0 iff each program P_i, with $i > 0$, is obtained by applying one of the rules: unfolding, temporal shift and folding, to P_{i-1}.

Example 3.7 (*Continued from example 3.5*). The final program in the transformation sequence is $P_{final} = \{2, 3, 4, 5, 6, 10, 19, 20, 21\}$, where:

(10) even_p(0).
(19) $\texttt{next}_0$ $\texttt{next}_0$ even_p(s(s(X))) $\leftarrow$ even_p(X).
(20) $\texttt{next}_1$ $\texttt{next}_0$ even_p(s(s(X))) $\leftarrow$ $\texttt{next}_0$ even_p(X).
(21) $\texttt{next}_1$ even_p(s(s(X))) $\leftarrow$ even_p(X).

(2) p(0).
(3) $\texttt{next}_0$ p(s(X)) $\leftarrow$ p(X).
(4) $\texttt{next}_1$ p(s(X)) $\leftarrow$ $\texttt{next}_0$ p(X).

(5) even(0).
(6) even(s(s(X))) $\leftarrow$ even(X).

It is easy to see that the set of clauses $\{10, 19, 20, 21\}$ in P_{final}, form a recursive definition for the predicate 'even_p'.

4 Correctness of the transformations

In this section, we present some correctness results concerning the transformation rules of section 3. For this, we define the notions of partial and total correctness.

Definition 4.1 (*Partially Correct Transformation*). Let $P_0, \ldots, P_i$ be a transformation sequence. We say that the transformation is *partially correct* iff $M(P_i) \subseteq M(P_0)$.

Definition 4.2 (*Totally Correct Transformation*). Let $P_0, \ldots, P_i$ be a transformation sequence. We say that the transformation is *totally correct* iff $M(P_i) = M(P_0)$.

The transformation rules presented in the previous section are partially correct:

Lemma 4.1 (**Partial Correctness**). *Let P_0,P_1, ..., P_s be a transformation sequence. If $M(P_0) = M(P_1) = \ldots = M(P_l)$, with $1 \leq l \leq s-1$, then $M(P_l) \supseteq M(P_{l+1})$.*

Lemma 4.1 shows that a transformation step is partially correct if all the preceding transformation steps are totally correct.

The transformation rules presented above are not totally correct unless we impose some additional restrictions on the application of the transformation rules. In order to introduce some sufficient conditions we need some more definitions.

Definition 4.3 (*Level of predicate/atom/clause*). Let P_0 be an initial program in a transformation sequence. To each predicate p in P_0, we assign a non negative integer l_p called the *level of* p. The *level of an atom* is the level of its predicate. The *level of a clause* is the level of its head.

Assumption 4.1 (**On the assignment of levels in the initial program**).

1. *All primitive predicates are assigned the level 0.*

2. *Each transformable predicate is assigned a level $i \geq 1$ such that for each clause C in P_0, the level of the head atom of C is greater than or equal to the level of each atom in the body of C.*

Definition 4.4 (*Unfolding state*). Let P_l, with $l \geq 0$, be a program in a transformation sequence, starting from P_0. To each clause C in P_l^t we assign a pair $[d_C, b_C]$ of natural numbers called the *unfolding state of* C, where d_C is called the *descent level of* C while b_C is called the *boundary unfolding number* (or *BU-number* for short) of C. The unfolding state of a clause is defined as follows:

1. If C is a clause in P_0^t and l_C is the level of C, then $[d_C, b_C] = [l_C, 1]$.
2. Let C be the result of unfolding a clause C' at a non primitive atom A, using a clause D. Let $[d_{C'}, b_{C'}]$ and $[d_D, b_D]$ be the unfolding states of C' and D respectively. Then the unfolding state of C is $[d_C, b_C]$, where $d_C = min\{d_{C'}, d_D\}$ and

$$b_C = \begin{cases} b_{C'} & \text{if } d_{C'} < d_D \\ b_D & \text{if } d_{C'} > d_D \\ b_{C'} + b_D & \text{if } d_{C'} = d_D \end{cases}$$

3. Let C be the result of applying the temporal shift rule to a clause C' or the result of unfolding C' at a primitive atom. Let U be the unfolding state of C'. Then the unfolding state of C is also U.
4. Let C be the result of folding a clause C' in P_{l-1}^t using a clause D in P_0^t. Let $[d_{C'}, b_{C'}]$ be the unfolding state of C and l_D be the level of D. Then the unfolding state of C is $[d_C, b_C]$, where $d_C = d_{C'}$ and

$$b_C = \begin{cases} b_{C'} & \text{if } d_C < l_D \\ b_{C'} - 1 & \text{if } d_C = l_D \end{cases}$$

Using the above definition, we introduce the following sufficient condition which ensure the correctness of the folding transformation rule.

Assumption 4.2 (**On the application of Folding**). *Let P_l, with $l \geq 0$, be a program in a transformation sequence starting from P_0 and P_{l+1} a program obtained by applying the folding rule to P_l. Let C and D be the folded and the folding clauses respectively. Let $[d_C, b_C]$ be the unfolding state of C, and let l_D be the level of D. The folding of C using D is said to be* valid *iff $l_D > d_C$ or ($l_D = d_C$ and $b_C > 1$).*

Theorem 4.1 (**Total Correctness**). *Let P_0 be an initial program in a transformation sequence, and P_l, with $l > 0$, a program obtained from P_0 by applying a sequence of transformation steps. Then, $M(P_l) = M(P_0)$.*

5 A continuation passing style transformation for Cactus programs

The transformation system presented in section 3 can be used to define a continuation passing style (CPS) transformation algorithm for Cactus programs. This algorithm is an extension of the algorithm presented in [ST89] for definite clause programs. As in [ST89] each argument of a program predicate is classified either as *input* or as *output* argument. For each program predicate[3] $\mathtt{p}(\overline{\mathtt{X}}, \overline{\mathtt{Y}})$ we introduce a corresponding continuation passing style

[3] $\overline{X}$ and $\overline{\mathtt{Y}}$ are mutually disjoint tuples of variables such that $\overline{X}$ corresponds to the arguments specified as input and $\overline{\mathtt{Y}}$ corresponds to the arguments specified as output.

predicate $\mathtt{p\_C}(\overline{\mathtt{X}},\mathtt{C})$ called a *closure predicate* pairing with an auxiliary predicate $\mathtt{cont\_p}(\overline{\mathtt{Y}},\mathtt{C})$ called a *continuation predicate*. Using these predicates we define a clause called an *existential continuation form* for $\mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}})$ as follows:

$$\mathtt{p\_C}(\overline{\mathtt{X}},\mathtt{C}) \leftarrow \mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}}), \mathtt{cont\_p}(\overline{\mathtt{Y}},\mathtt{C})$$

The variable `C` is called the *continuation variable*. Using these notions we can now define the CPS algorithm. The input of the algorithm is a Cactus program P while its output is a program P_{CPS} in continuation passing style form.

CPS transformation algorithm:

Step 1: Let Def_e be the set of existential continuation forms for the predicates in P.
Step 2: For every clause (Cl) in Def_e:

$$(Cl) \qquad \mathtt{p\_C}(\overline{\mathtt{X}},\mathtt{C}) \leftarrow \mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}}), \mathtt{cont\_p}(\overline{\mathtt{Y}},\mathtt{C})$$

apply the following process. Unfold (Cl) at $\mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}})$ obtaining a set of clauses of the form:

$$(Cl_i) \qquad \mathtt{p\_C}(\mathtt{Si},\mathtt{C}) \leftarrow \mathtt{E_i}, \mathtt{cont\_p}(\mathtt{Ti},\mathtt{C})$$

with $1 \leq i \leq n$, where n is the number of program clauses whose head is unifiable with $\mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}})$, and $\mathtt{S_i}$, $\mathtt{T_i}$ are instances of $\overline{\mathtt{X}}$ and $\overline{\mathtt{Y}}$ respectively.
For each $i : 1 \leq i \leq n$, do
Case 1: If E_i is empty add (Cl_i) to P_{CPS}.
Case 2: If E_i is non empty, write (Cl_i) as

$$(Cl_i) \qquad \mathtt{p\_C}(\mathtt{S_i},\mathtt{C}) \leftarrow \mathtt{R}\ \mathtt{q}(\mathtt{S},\mathtt{T}), \mathtt{F_i}, \mathtt{cont\_p}(\mathtt{T_i},\mathtt{C})$$

where R is a temporal reference and `q(S,T)` is a classical atom. Then introduce and add to Def_c a new clause, called a *continuation definition*:

$$(D_j) \qquad \mathtt{R}\ \mathtt{cont\_q}(\mathtt{T},\mathtt{f}(\overline{\mathtt{W}},\mathtt{C})) \leftarrow \mathtt{F_i}, \mathtt{cont\_p}(\mathtt{T_i},\mathtt{C})$$

where $\overline{\mathtt{W}}$ is a tuple of variables such that $\overline{\mathtt{W}} = FreeVars(\mathtt{S_i},\mathtt{S}) \cap FreeVars(\mathtt{T},\mathtt{F_i},\mathtt{T_i})$ and `f` is a fresh function symbol. Fold (Cl_i) using by the definition (D_j) to get

$$(Cl_i') \qquad \mathtt{p\_C}(\mathtt{Si},\mathtt{C}) \leftarrow \mathtt{R}\ \mathtt{q}(\mathtt{S},\mathtt{T}), \mathtt{R}\ \mathtt{cont\_q}(\mathtt{T},\mathtt{f}(\overline{\mathtt{W}},\mathtt{C}))$$

Fold further (Cl_i') using the existential continuation form for the predicate `q` to get:

$$(Cl_i'') \qquad \mathtt{p\_C}(\mathtt{Si},\mathtt{C}) \leftarrow \mathtt{R}\ \mathtt{q\_C}(\mathtt{S},\mathtt{f}(\overline{\mathtt{W}},\mathtt{C}))$$

Add (Cl_i'') to P_{CPS}.
Step 3: For every clause (D_i) in Def_c

$$(D_i) \qquad \mathtt{R}\ \mathtt{cont\_q}(\mathtt{T},\mathtt{f}(\overline{\mathtt{W}},\mathtt{C})) \leftarrow \mathtt{F_i}, \mathtt{cont\_p}(\mathtt{T_i},\mathtt{C})$$

apply the following process.
Case 1: If F_i is empty, add (D_i) to P_{CPS}.
Case 2: If F_i is non empty, transform (D_i) following exactly the Case 2 in step 2.
Step 4: Supply P_{CPS} with a set of unit clauses, called *terminators*, constructed as follows: For each program predicate $\mathtt{p}(\overline{\mathtt{X}},\overline{\mathtt{Y}})$ in P, add a unit clause of the form $\mathtt{cont\_p}(\overline{\mathtt{Y}},\mathtt{f_0^p}(\overline{\mathtt{Y}}))$.

Example 5.1. Let $P_0 = \{1, 2, 3, 4, 5, 6\}$ be the following Cactus program:

(1) $\texttt{first num(0).}$
(2) $\texttt{next}_0\ \texttt{num(s(X))} \leftarrow \texttt{num(X).}$
(3) $\texttt{next}_1\ \texttt{num(s(s(X)))} \leftarrow \texttt{num(X).}$
(4) $\texttt{next}_2\ \texttt{num(X)} \leftarrow \texttt{next}_0\ \texttt{num(X0)}, \texttt{next}_1\ \texttt{num(X1)}, \texttt{sum(X0,X1,X).}$
(5) $\texttt{sum(0,Y,Y).}$
(6) $\texttt{sum(s(X),Y,s(Z))} \leftarrow \texttt{sum(X,Y,Z).}$

Suppose that the patterns `num(+)` and `sum(+,+,-)`, reflect the classification of the arguments of the program predicates as input (denoted by '`+`') or output (denoted by '`-`').
By applying the step 1 of the CPS algorithm we get the set $Def_e = \{D1, D2\}$, where:

($D1$) $\texttt{numC(X,C)} \leftarrow \texttt{num(X)}, \texttt{cont\_num(C).}$
($D2$) $\texttt{sumC(X,Y,C)} \leftarrow \texttt{sum(X,Y,Z)}, \texttt{cont\_sum(Z,C).}$

Now we proceed in step 2 and unfold (D1) at `num(X)`. We get

(7) $\texttt{first numC(0,C)} \leftarrow \texttt{first cont\_num(C).}$
(8) $\texttt{next}_0\ \texttt{numC(s(X),C)} \leftarrow \texttt{num(X)}, \texttt{next}_0\ \texttt{cont\_num(C).}$
(9) $\texttt{next}_1\ \texttt{numC(s(s(X)),C)} \leftarrow \texttt{num(X)}, \texttt{next}_1\ \texttt{cont\_num(C).}$
(10) $\texttt{next}_2\ \texttt{numC(X,C)} \leftarrow \texttt{next}_0\ \texttt{num(X0)}, \texttt{next}_1\ \texttt{num(X1)},$
$\texttt{sum(X0,X1,X)}, \texttt{next}_2\ \texttt{cont\_num(C).}$

We add (7) to P_{CPS}. For clause (8) we introduce:

($D3$) $\texttt{cont\_num(f}_1\texttt{(C))} \leftarrow \texttt{next}_0\ \texttt{cont\_num(C).}$

Folding (8) using (D3) we get

(11) $\texttt{next}_0\ \texttt{numC(s(X),C)} \leftarrow \texttt{num(X)}, \texttt{cont\_num(f}_1\texttt{(C)).}$

Folding (11) using (D1) we get clause (12) and add it to P_{CPS}:

(12) $\texttt{next}_0\ \texttt{numC(s(X),C)} \leftarrow \texttt{numC(X,f}_1\texttt{(C)).}$

For clause (9) we introduce:

($D4$) $\texttt{cont\_num(f}_2\texttt{(C))} \leftarrow \texttt{next}_1\ \texttt{cont\_num(C).}$

Folding (9) using (D4) we get:

(13) $\texttt{next}_1\ \texttt{numC(s(s(X)),C)} \leftarrow \texttt{num(X)}, \texttt{cont\_num(f}_2\texttt{(C)).}$

Folding (13) using (D1) we get clause (14)which is added to P_{CPS}:

(14) $\texttt{next}_1\ \texttt{numC(s(s(X)),C)} \leftarrow \texttt{numC(X,f}_2\texttt{(C)).}$

For (10) we introduce:

($D5$) $\texttt{next}_0\ \texttt{cont\_num(f}_3\texttt{(X,X0,C))} \leftarrow \texttt{next}_1\ \texttt{num(X1)},$
$\texttt{sum(X0,X1,X)}, \texttt{next}_2\ \texttt{cont\_num(C).}$

Folding (10) using (D5) we get:

(15) $\texttt{next}_2\ \texttt{numC(X,C)} \leftarrow \texttt{next}_0\ \texttt{num(X0)}, \texttt{next}_0\ \texttt{cont\_num(f}_3\texttt{(X,X0,C)).}$

Folding (15) using (D1) we get clause (16) which is added to P_{CPS}:

(16) $\texttt{next}_2\ \texttt{numC(X,C)} \leftarrow \texttt{next}_0\ \texttt{numC(X0,f}_3\texttt{(X,X0,C)).}$

Now we unfold (D2) at '`sum(X,Y,Z)`'. We get

$$\begin{array}{lll}(17) & \mathtt{sumC(0,Y,C)} \leftarrow \mathtt{cont\_sum(Y,C)}. \\ (18) & \mathtt{sumC(s(X),Y,C)} \leftarrow \mathtt{sum(X,Y,Z),cont\_sum(s(Z),C)}.\end{array}$$

(17) is added to P_{CPS}. For (18) we introduce clause (D6):

$$(D6) \quad \mathtt{cont\_sum(Z,f_4(C))} \leftarrow \mathtt{cont\_sum(s(Z),C)}.$$

Folding (18) using (D6) we get

$$(19) \quad \mathtt{sumC(s(X),Y,C)} \leftarrow \mathtt{sum(X,Y,Z),cont\_sum(Z,f_4(C))}.$$

Folding (19) using (D2) we get (20) which is added to P_{CPS}:

$$(20) \quad \mathtt{sumC(s(X),Y,C)} \leftarrow \mathtt{sumC(X,Y,f_4(C))}.$$

Going into step 3 we add (D3), (D4) and (D6) to P_{CPS}. For (D5) we introduce:

$$(D7) \quad \mathtt{next_1\ cont\_num(f_5(X,X0,X1,C))} \leftarrow \mathtt{sum(X0,X1,X),next_2\ cont\_num(C)}.$$

Folding (D5) using (D7) we get:

$$\begin{array}{ll}(21) & \mathtt{next_0\ cont\_num(f_3(X,X0,C))} \leftarrow \mathtt{next_1\ num(X1)}, \\ & \quad \mathtt{next_1\ cont\_num(f_5(X,X0,X1,C))}.\end{array}$$

Folding (21) using (D1) we get clause (22) which is added to P_{CPS}:

$$(22) \quad \mathtt{next_0\ cont\_num(f_3(X,X0,C))} \leftarrow \mathtt{next_1\ numC(X1,f_5(X,X0,X1,C))}.$$

For (D7) we introduce (D8) which is also added to P_{CPS}:

$$(D8) \quad \mathtt{cont\_sum(X,f_6(X,C))} \leftarrow \mathtt{next_2\ cont\_num(C)}.$$

Folding (D7) using (D8) we get

$$(23) \quad \mathtt{next_1\ cont\_num(f_5(X,X0,X1,C))} \leftarrow \mathtt{sum(X0,X1,X),cont\_sum(X,f_6(X,C))}.$$

Folding (23) using (D2) we get (24) which is added to P_{CPS}:

$$(24) \quad \mathtt{next_1\ cont\_num(f_5(X,X0,X1,C))} \leftarrow \mathtt{sumC(X0,X1,f_6(X,C))}.$$

Now applying step 4, we add the terminators:

$$\begin{array}{ll}(25) & \mathtt{cont\_num(f_0^{num})}. \\ (26) & \mathtt{cont\_sum(Y,f_0^{sum}(Y))}.\end{array}$$

Collecting together all clauses in P_{CPS} we obtain:

$$\begin{array}{ll}(7) & \mathtt{first\ numC(0,C)} \leftarrow \mathtt{first\ cont\_num(C)}. \\ (12) & \mathtt{next_0\ numC(s(X),C)} \leftarrow \mathtt{numC(X,f_1(C))}. \\ (14) & \mathtt{next_1\ numC(s(s(X)),C)} \leftarrow \mathtt{numC(X,f_2(C))}. \\ (16) & \mathtt{next_2\ numC(X,C)} \leftarrow \mathtt{next_0\ numC(X0,f_3(X,X0,C))}. \\ (25) & \mathtt{cont\_num(f_0^{num})}. \\ (D3) & \mathtt{cont\_num(f_1(C))} \leftarrow \mathtt{next_0\ cont\_num(C)}. \\ (D4) & \mathtt{cont\_num(f_2(C))} \leftarrow \mathtt{next_1\ cont\_num(C)}. \\ (22) & \mathtt{next_0\ cont\_num(f_3(X,X0,C))} \leftarrow \mathtt{next_1\ numC(X1,f_5(X,X0,X1,C))}. \\ (24) & \mathtt{next_1\ cont\_num(f_5(X,X0,X1,C))} \leftarrow \mathtt{sumC(X0,X1,f_6(X,C))}. \\ (17) & \mathtt{sumC(0,Y,C)} \leftarrow \mathtt{cont\_sum(Y,C)}. \\ (20) & \mathtt{sumC(s(X),Y,C)} \leftarrow \mathtt{sumC(X,Y,f_4(C))}.\end{array}$$

(26) $\quad$ `cont_sum(Y, f_0^sum(Y)).`
($D6$) $\quad$ `cont_sum(Z, f_4(C)) ← cont_sum(s(Z), C).`
($D8$) $\quad$ `cont_sum(X, f_6(X, C)) ← next_2 cont_num(C).`

Although the CPS algorithm for definite clause programs presented in [ST89] applies to every definite clause program, our algorithm does not apply to every Cactus program. This is due to the restrictions imposed by the definitions of the unfolding and folding transformation rules. Nevertheless, we can show that the algorithm applies to a wide class of Cactus programs. In order to define this class, we will define the notion of '*naughty clauses*'.

Definition 5.1 (*Naughty clause*). A clause C is said to be a *naughty clause* if there is a canonical atom in the body of C and there is at least one non canonical atom either in the head or in the body of C.

It is easy to see that our algorithm applies to every Cactus program not containing naughty clauses as in this case the application of the unfolding and folding rules in the algorithm do not violate the restrictions imposed by the corresponding definitions.

Theorem 5.1 (**Correctness of the CPS algorithm**). *Let P_0 be a Cactus program not containing naughty clauses and P_{CPS} be the program obtained by applying the CPS algorithm to P_0. Then, for every ground temporal atom '$p(a,b)$' in the Herbrand base of P_0 where 'a' is a tuple corresponding to the arguments of 'p' specified as input and 'b' the tuple corresponding to the arguments of 'p' specified as output, $P_0 \models p(a,b)$ iff $P_{CPS} \models p\_C(a, f_0^p(b))$.*

6 Discussion

Program transformation techniques have been widely used in definite clause logic programming as well as in functional programming. This is why we believe that they will also be proved useful in temporal logic programming languages as well.

In this paper we define a program transformation system for branching-time (Cactus) logic programs and present sufficient conditions for their correctness.

As an application of the transformation system that we propose, we develop an algorithm which can be used to transform a wide class of branching time logic programs into equivalent programs in continuation passing style form. A program obtained by applying the CPS algorithm has a special form i.e. no clause in this program has more than one atoms it its body.

Both the transformation system and the CPS algorithm apply also to Chronolog and Chronolog($\mathcal{Z}$) programs [Wad88, OWD93]. Moreover, it is easy to adapt them to apply to multidimensional logic programming languages [OD94] as well.

As in transformation of definite clause programs, transformation strategies are needed to guide the application of the transformation rules. Nevertheless, it seems straightforward to extend the transformation strategies developed in the context of definite clause program transformation [PP95, PP93, PP94b] for the case of branching time logic programs.

It is important to note here that (zero-order) branching-time and multidimensional functional languages have been recognized [RW97, Yag84, Ron94] as appropriate target languages for transforming first-order functional programs. This transformation is very useful since zero-order branching time programs can be efficiently executed using tagged, demand-driven evaluation [FW87]. We believe that the transformation system that we propose in this paper may be used as a tool in order to define a similar transformation from definite clause programs into multidimensional logic programs.

References

[AGK96] F. Afrati, M. Gergatsoulis, and M. Katzouraki. On transformations into linear database logic programs. In D. Bjørner, M. Broy, and I. Pottosin, editors, *Perspectives of Systems Informatics (PSI'96), Proceedings*, Lecture Notes in Computer Science (LNCS) 1181, pages 433–444. Springer-Verlag, 1996.

[Bau93] M. Baudinet. A simple proof of the completeness of temporal logic programming. In L. Farinas del Cerro and M. Penttonen, editors, *Intensional Logics for Programming*, pages 51–83. Oxford University Press, 1993.

[BC93] A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer substitutions of logic programs. *The Journal of Logic Programming*, 16(1 & 2):47–87, May 1993.

[BCD90] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J.ACM*, 24(1):44–67, Jan. 1977.

[Brz91] C. Brzoska. Temporal logic programming and its relation to constraint logic programming. In *Proc. of the Logic Programming Symposium*, pages 661–677. MIT Press, 1991.

[Brz93] C. Brzoska. Temporal logic programming with bounded universal modality goals. In D. S. Warren, editor, *Proc. of the Tenth International Conference on Logic Programming*, pages 239–256. MIT Press, 1993.

[FW87] A. Faustini and W. Wadge. An Eductive Interpreter for the Language pLucid. In *Proceedings of the SIGPLAN 87 Conference on Interpreters and Interpretive Techniques (SIGPLAN Notices 22(7))*, pages 86–91, 1987.

[Ger94] M. Gergatsoulis. *Logic program transformations: Transformation rules and application strategies.* PhD thesis, Dept. of Computer Science, University of Athens, 1994. (In Greek).

[GK94] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'94), Proceedings*, Lecture Notes in Computer Science (LNCS) 844, pages 340–354. Springer-Verlag, 1994.

[GRP96] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Disjunctive Chronolog. In M. Chacravarty, Y. Guo, and T. Ida, editors, *Proceedings of the JICSLP'96 Post-Conference Workshop "Multi-Paradigm Logic Programming"*, pages 129–136, Bonn, 5-6 Sept. 1996.

[GRP97] M. Gergatsoulis, P. Rondogiannis, and T. Panayiotopoulos. Proof procedures for branching-time logic programs. In W. W. Wadge, editor, *Proc. of the Tenth International Symposium on Languages for Intensional Programming (ISLIP'97), May 15-17, Victoria BC, Canada*, pages 12–26, 1997.

[Hry93] T. Hrycej. A temporal extension of Prolog. *The Journal of Logic Programming*, 15:113–145, 1993.

[KK90] T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformations. *Theoretical Computer Science*, 75:139–156, 1990.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Logic Programming*, 11(3 & 4):217–242, Oct./Nov. 1991.

[OD94] M. A. Orgun and W. Du. Multi-dimensional logic programming. *Journal of Computing and Information*, 1(1):1501–1520, 1994. Special Issue: Proc. of the 6th International Conf. on Computing and Information.

[OM94] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In *Proc. of the First International Conference on Temporal Logics (ICTL'94)*, Lecture Notes in Computer Science (LNCS) 827, pages 445–479. Springer-Verlag, 1994.

[Org91] M. A. Orgun. *Intensional logic programming*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1991.

[OWD93] M. A. Orgun, W. W. Wadge, and W. Du. Chronolog($\mathcal{Z}$): Linear-time logic programming. In O. Abou-Rabia, C. K. Chang, and W. W. Koczkodaj, editors, *Proc. of the Fifth International Conference on Computing and Information*, pages 545–549. IEEE Computer Society Press, 1993.

[PP93] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.

[PP94a] A. Pettorossi and M. Proietti. Transformation of logic programs : Foundations and techniques. *The Journal of Logic Programming*, 19/20:261–320, May/July 1994.

[PP94b] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Yves Deville, editor, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 141–158. Springer-Verlag, 1994.

[PP95] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.

[PP97] A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press, 1997.

[RGP97a] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. The branching-time logic programming language Cactus and its applications. Technical Report 3-97, Dept. of Computer Science, University of Ioannina, Greece, 1997.

[RGP97b] P. Rondogiannis, M. Gergatsoulis, and T. Panayiotopoulos. *Cactus*: A branching-time logic programming language. In *Proc. of the First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97, Bad Honnef, Germany*, Lecture Notes in Artificial Intelligence (LNAI) 1244, pages 511–524. Springer, June 1997.

[Ron94] P. Rondogiannis. *Higher-order functional languages and intensional logic*. PhD thesis, Dept. of Computer Science, University of Victoria, Canada, December 1994.

[RW97] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[Sek93] H. Seki. Unfold/fold transformations for general logic programs for the well-founded semantics. *The Journal of Logic Programming*, 16(1 & 2):5–23, May 1993.

[ST84] T. Sato and H. Tamaki. Transformational logic program synthesis. In *International Conference on Fifth Generation Computer Systems*, pages 195–201, 1984.

[ST89] T. Sato and H. Tamaki. Existential continuation. *New Generation Computing*, 6:421–438, 1989.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.

[TS86] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report No 86-4, Dept. of Information Science, Ibaraki University, Japan, 1986.

[Wad88] W. W. Wadge. Tense logic programming: A respectable alternative. In *Proc. of the 1988 International Symposium on Lucid and Intensional Programming*, pages 26–32, 1988.

[Yag84] A. Yaghi. *The intensional implementation technique for functional languages*. PhD thesis, Dept. of Computer Science, University of Warwick, Coventry, UK, 1984.