# Alternating Fixed Point Logic Programs

Witold Charatonik *† David McAllester ‡ Damian Niwiński §¶
Andreas Podelski * Igor Walukiewicz Σ

May 8, 1998

## Abstract

An alternating fixed point (afp) logic program may contain arbitrary nestings of least and greatest fixed points. Afp logic programs can naturally and exactly express any safety or liveness properties for any (infinite state) reactive system whose state transition relation is recursively enumerable. We show that in spite of this extreme expressive power, nontrivial static analysis is possible. In particular, we show that a variety of set-based type inference algorithms for logic programs generalize to the afp case. In the afp case the infered types are regular in the sense that they are definable by Rabin tree automata. In a different reading, our results express a kind of robustness of the notion of regularity: alternating Rabin tree automata preserve the same expressiveness and algorithmic complexity if we extend them with pushdown transition rules (in the same way Büchi extended word automata to canonical systems).

## 1 Introduction

It well known that logic programs can be assigned either a least fixed point or a greatest fixed point semantics. Under the more common least fixed point semantics an atom is true in the semantics if there exists an a finite derivation of that atom using the clauses of the program. Under the greatest fixed piont semantics an atom is true if there exists any (possibly infinite) derivation of that atom. Here we consider logic programs in which each predicate is assigned a priority and an atom is true if there exists a derivation satsisfying a parity condition, i.e., there exists a derivation such that on every infinite path in the derivation the maximum priority occuring infinitely often is even. In this semantics we allow infinite as well as finite terms. Using standard techniques we show that the parity condition can be viewed as a nesting of least and greatest fixed points — predicates of odd priority can be interpreted as having least fixed point definitions and predicates of even priority as having greatest fixed point definitions. Predicates of lower priority can be viewed as being defined inside the body of the fixed point definition of predicates of higher priority. Logic programs with parity conditions corresponds to an alternation of least and greatest fixed points — we call them alternating fixed point (afp) programs.

Afp logic programs are extremely expressive. They can naturally and exactly express any safety or liveness property of any (infinite state) reactive systems whose state transition relation is recursively enumerable. In spite of the extreme expressive power of afp logic programs we show here that nontrivial static anylsis of these programs is possible — standard methods of analyzing least fixed point logic programs can be generalized to the afp case.

A variety of set-based type inference algorithms infer regular data types for program variables [?, ?, ?, ?, ?]. Regularity is a fundamental property of sets of words and trees. In the afp case it is natural to allow infinite data structures and to consider data types defined by Rabin tree automata. In this paper we construct a synthesis of set-

*Max-Planck-Institut für Informatik
D-66123 Saarbrücken, {witold;podelski}@mpi-sb.mpg.de
†on leave from Wroclaw University
‡AT&T Labs
Murray Hill NJ 07974, dmac@research.att.com
§Institute of Informatics, Warsaw University 02-097 Warsaw, Poland, {niwinski,igw}@mimuw.edu.pl
¶visiting Max-Planck-Institut

based program analysis and Rabin tree automata — a type inference algorithm where each infered type is defined by a Rabin automata over infinite trees.

We will use the notion of a uniform logic program as a central tool for program analysis. A uniform program is one in which all clauses have one of the following three forms.

$$p(f(x_1, \ \ldots, \ x_n)) \leftarrow q_1(x_1), \ \ldots, \ q_n(x_n) \qquad (1)$$

$$p(x) \leftarrow q_1(x), \ \ldots, \ q_n(x) \qquad (2)$$

$$p(x) \leftarrow q(t) \qquad (3)$$

Clauses of form 1, 2, and 3 will be called regular, alternation, and projection clauses respectively.

An afp program will be called regular if it consists entirely of regular clauses. A regular afp program is easily seen to be equivalent to a tree automata over infinite trees with a parity condition. It is known that the sets definable by automata with parity conditions coincide with the sets definable by Rabin tree automata. A set of (possibly infinite) terms will be called regular if it is definable by a regular afp logic program, or equivalently, definable by a Rabin tree automata.

An afp program will be called alternating-regular if it consists entirely of regular and alternation clauses. Alternating-regular afp programs can be viewed as alternating tree automata over infinite trees with parity conditions. Such automata are essentially equivalent to the modal $\mu$-calculus. It is known that these automata define regular sets and that the emptyness problem for these automata is EXPTIME complete.

As stated above, an afp program consisting of regular, alternation, and projection clauses is called uniform. Under least fixed point semantics it is known that the sets of finite trees definable by uniform programs are regular and that emptiness for least fixed point uniform programs is EXPTIME-complete [**?**]. Our two main results are that uniform afp programs define regular sets and that emptyness for uniform afp programs is EXPTIME complete.

Standard relaxations from an arbitrary program to a uniform program easily generalize to the afp case — here we give the generalization of the relaxation introduced by Heintze and Jaffar [**?**]. In the afp case the relaxation maps an arbitrary afp program to a uniform afp program. Our two main results provide an effective procedure for "solving" for the meaning of the predicates in a uniform afp program, i.e., for translating to an equivalent purely regular afp program.

A different reading of our results stems from the connection between set constraints and Büchi's canonical systems [**?**], which was observed first in [**?**]. Canonical systems may be viewed as a kind of pushdown automata whose input string is the initial stack contents. In this view, standard non-deterministic automata (nfa's) correspond to the special case where only pop transitions are allowed. Uniform programs generalize canonical systems to acceptors of infinite trees with alternation and with stacks containing infinite trees[1]; they extend alternating Rabin automata in the same sense that canonical systems extend nfa's. Our result thus expresses a kind of robustness of the notion of regularity wrt. expressiveness and algorithmic complexity under this extension.

Finite automata with a stack have raised interest as a kind of infinite reactive systems (then also called pushdown processes) for which various properties are decidable [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], and also in relation with dynamic logics (see, e.g., [**?**]). Uniform programs can easily represent pushdown processes. The clause $p(a(x)) \leftarrow q(x)$ expresses a *pop* transition from state $p$ to state $q$ under the condition that the first letter of the stack is $a$; the clause $p(x) \leftarrow q(x)$ is an $\varepsilon$-transition leaving the stack $x$ invariant; the clause $p(x) \leftarrow q(a(x))$ is a *push* transition.[2] The computation trees (*pushdown trees* of [**?**]) become logic program derivations. The emptyness test for uniform afp programs yields a full test of the $\mu$-calculus property $\varphi$ (in comparison, the method described in [**?**] applies only to CTL properties). This still holds true if we extend pushdown processes to the form that corresponds exactly to uniform programs, i.e., with tree-like stacks, non-deterministic guesses of stack contents and alternation. The issue of an *input* is absent in [**?**] (or, an input tree is always the starting stack symbol). The uniform programs extend pushdown processes in several directions.

[1] The tree-like stack distinguishes our notion of automaton of from the ones studied in [**?**, **?**] which have a standard stack (and also a less general acceptance condition).

[2] These three kinds of clauses correspond to the three kinds of rules in canonical systems, which Büchi called, respectively, contraction, neutration and expansion [**?**].

They take nontrivial inputs, permit free variables in bodies of clauses, and allow functional symbols of arbitrary arity.

Our two main results require proving a form of equivalence between a given uniform afp program and a constructed alternating-regular afp program. To prove that the two afp programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent we translate $\mathcal{P}$-derivations to $\mathcal{P}'$-derivations and vice versa. Such translations can be done by a process of local rewriting — each use of an inference rule in $\mathcal{P}'$ is translated to a corresponding ue of an inference rule in $\mathcal{P}$. However, for afp logic programs derivations are generally infinite. The translation is defined as the limit of an infinite rewriting process. The main difficulty is proving that the infinite rewriting preserves parity. Before proving the main results we prove a parity subject reduction theorem stating that, under a fairly general notion of reduction, parity is preserved by transfinite reduction.

Although we prove the main theorems from our parity subject reduction result, the same theorems can be proved using game-theoretic techniques in a way that avoids parity subject reduction. We need to show an equivalence between two afp logic programs $\mathcal{P}$ and $\mathcal{P}'$. It turns out that parity is only a serious issue in one direction of the translation, e.g., is easy to prove that $\mathcal{P}$ to $\mathcal{P}'$ translation preserves parity but hard to prove that the $\mathcal{P}'$ to $\mathcal{P}$ translation preserves parity. Using game-theoretic techniques it is possible to avoid the $\mathcal{P}'$ to $\mathcal{P}$ translation entirely. Rather than translation $\mathcal{P}'$-derivations to $\mathcal{P}$-derivations one can translate $\mathcal{P}$-coderivations to $\mathcal{P}'$-coderivations. A coderivation is a kind of proof that a certain atom is *not* derivable. Using game-theoretic determinancy for parity games it is possible to prove that for any atom $\Phi$ and afp program $\mathcal{P}$ there is either a parity-satsifying $\mathcal{P}$-derivation of $\Phi$ or a parity-satsifying $\mathcal{P}$-coderivation of $\neg\Phi$. So to prove that $\mathcal{P}$ is equivalent to $\mathcal{P}'$ it suffices to give a method of translating both derivations and coderivations from $\mathcal{P}$ to $\mathcal{P}'$. In our application it is easy to show that translations of both derivations and coderivations from $\mathcal{P}$ to $\mathcal{P}'$ preserve parity.

There are two reasons for giving a parity subject reduction proof rather than a derivation-coderivation proof. First, parity subject reduction seems novel and may have other applications. Second, if ignoring parity the translation from $\mathcal{P}$-derivations to $\mathcal{P}'$-derivations is significantly more complex than the translation from $\mathcal{P}'$-derivations to $\mathcal{P}$-derivations. The translation of $\mathcal{P}$-codervations to $\mathcal{P}'$-coderivations is even more complex. The parity subject reduction theorem allows the use of the simpler $\mathcal{P}'$ to $\mathcal{P}$ translation and simplifies the overall proof.

## 2 Programs, priorities, fixed points

We define a *logic program* to be a finite set of Horn clauses, i.e., a set of expressions of the form $H \leftarrow B$ where $H$ is a first order atom, i.e., a predicate applied to first order terms, and $B$ is a finite set of first order atoms. It is well known that logic programs can be assigned both a least and a greatest fixed point semantics. In this section we will use an assignment of priorities to predicate symbols to define a semantics that combines least and greatest fixed points. First, however, it is useful to review the standard least and greatest fixed point semantics of logic programs.

In greatest fixed point semantics one usually works with infinite terms, e.g., the infinite term $f(f(f(\ldots)))$ where $f$ is a monadic function symbol. Throughout the remainder of this paper we will use the phrase "ground term" to mean a possibly infinite term which does not contain variables. A ground atom is defined to be a predicate symbol of $n$ arguments applied to $n$ (possibly infinite) ground terms. A ground clause is an expression of the form $H \leftarrow B$ where $H$ is a ground atom and $B$ is a finite set of ground atoms. We say that a ground clause $H' \leftarrow B'$ is a ground instance of a Horn clause $H \leftarrow B$ if there exists a substitution $\sigma$ mapping variables to ground terms such that $H' \leftarrow B'$ is the result of replacing each variable $x$ in $H \leftarrow B$ by $\sigma(x)$. Note that under these definitions a Horn clause contains only finite terms while a ground instance of that clause may contain infinite terms.

Consider a possibly infinite tree $r$ such that each node of $w$ of $r$ is labeled with a ground atom denoted $r(w)$. We let $C(w)$ denote the set of nodes which are children of the node $w$ and let $r(C(w))$ denote the set of ground atoms of the form $r(s)$ for $s \in C(w)$. Given a logic program $\mathcal{P}$ we say that $r$ is a $\mathcal{P}$-derivation if for each node $w$ in $r$ we have that $r(w) \leftarrow r(C(w))$ is a ground instance of a clause in $\mathcal{P}$. For any logic program $\mathcal{P}$ we define the greatest fixed point meaning of $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]_\nu$, to be the set of all ground atoms $\Phi$ such that there exists a (possibly infinite) derivation tree whose root node is labeled with

$\Phi$. We define the least fixed point semantics of $\mathcal{P}$, denoted $[\![\mathcal{P}]\!]_\mu$, to be the set of ground atoms such that there exists a finite $\mathcal{P}$-derivation whose root is labeled with $\Phi$. Note that if a program $\mathcal{P}$ contains a clause of the form $p(x)$ (with an empty right hand side) then $[\![\mathcal{P}]\!]_\mu$ contains infinite atoms — the least fixed point semantics of $\mathcal{P}$ over infinite atoms is different from the least fixed point meaning of $\mathcal{P}$ over only finite atoms. However, for any finite signature finiteness is definable, i.e., one can write clauses defining a monadic predicate $q$ such that $q(t) \in [\![\mathcal{P}]\!]_\mu$ if and only if $t$ is a finite term. Since finiteness is definable (under least fixed point semantics) interpretation over infinite terms subsumes interpretation over finite terms.

**Priorities** We now generalize the semantics to allow arbitrary nestings of greatest and least fixed points. We define a alternating fixed point (afp) program to be a logic program in which every predicate symbol $p$ is associated with a non-negative integer $\Omega(p)$ called the priority of $p$. We say that a priority occurs in a program $\mathcal{P}$ if it equals $\Omega(p)$ for some $p$ occurring in $\mathcal{P}$. The priority of a ground atom is defined to be the priority of the predicate used in that atom. If $\mathcal{P}$ is an afp program then a $\mathcal{P}$-derivation is defined as before. Consider a $\mathcal{P}$-derivation $r$ and let $w$ be an infinite sequence $w_0 w_1 w_2 \ldots$ of nodes in a $r$ such that $w_{i+1}$ is a child of $w_i$. We will call such a sequence $w$ an infinite path in $r$. Let $\mathrm{Inf}(w)$ be the set of priorities occurring infinitely often on the path, i.e., the set of priorities $k$ such that there are infinitely many $w_i$ where $k$ is the priority of $r(w_i)$. We say that a path $w$ in $r$ satisfies the parity condition if the largest element of $\mathrm{Inf}(w)$, is even. A $\mathcal{P}$-derivation $r$ satisfies parity if every infinite path in $r$ satisfies parity. For any afp program $\mathcal{P}$ we define $[\![\mathcal{P}]\!]$ to be the set of ground atomic formulas $\Phi$ such that there exists an parity-satisfying $\mathcal{P}$-derivation where the root node is labeled with $\Phi$.

Note that if all predicates have priority 0 then $[\![\mathcal{P}]\!] = [\![\mathcal{P}]\!]_\nu$. On the other hand, if all predicates have priority 1, then $[\![\mathcal{P}]\!] = [\![\mathcal{P}]\!]_\mu$. As an example of the added power of mixed priorities consider the program $\mathcal{P}$ consisting of the following clauses where $p$ has priority 1 and $q$ has priority 2.

$$p(f(x)) \leftarrow p(x)$$

$$p(x) \leftarrow q(x)$$

$$q(g(x)) \leftarrow p(x)$$

Note that any infinite path in an parity-satisfying derivation must have an infinite number of occurrences of the predicate $q$. This implies that $[\![\mathcal{P}]\!]$ contains all atoms of the form $p(t)$ and $q(g(t))$ where $t$ is an infinite term constructed from the monadic functions $f$ and $g$ containing an infinite number of occurrences of $g$. Intuitively, since the predicate $p$ has an odd index it must "terminate" in the use of a higher priority predicate. Since the higher priority $q$ has an even index it need not terminate and can call itself — through terminating calls to $p$ — an infinite number of times. This program expresses the statement that from any point one will eventually encounter a $g$ — a kind of liveness property. It is possible to show that this property is not expressible under either purely least fixed point or purely greatest fixed point semantics.

**Fixed point semantics** We now give an alternate characterization of the semantics $[\![\mathcal{P}]\!]$ of a program $\mathcal{P}$ in terms of nested fixed point expressions. The fixed point characterization of $[\![\mathcal{P}]\!]$ relates the parity condition to nested alternations of least and greatest fixed points.

First we define the familiar $T_\mathcal{P}$ operator of logic programming. We define a database to be a set of ground atoms. For any database $A$ we define $T_\mathcal{P}(A)$ in the standard way to be the set of ground atoms $\Phi$ such that there exists a ground instance $\Phi \leftarrow B$ of a clause in $\mathcal{P}$ such that $B \subseteq A$. For any databases $A$ and $B$ and non-negative priority $k$ we define $A[k := B]$ to be the database such that $\Phi \in A[k := B]$ if either the priority of $\Phi$ is different from $k$ and $\Phi \in A$ or the priority of $\Phi$ equals $k$ and $\Phi \in B$. We now define the operator $T^k_\mathcal{P}$ such that $T^k_\mathcal{P}(A)$ equals $A[k := T_\mathcal{P}(A)]$. The operator $T^k_\mathcal{P}$ is analogous to $T_\mathcal{P}$ except that $T^k_\mathcal{P}$ only updates predicates of priority $k$. We now take fixed points of the operators $T^k_\mathcal{P}$ separately for each value of $k$. More specifically, for any integer $k$ we define $F^k_\mathcal{P}(A)$ as follows. For negative $k$ we define $F^k_\mathcal{P}(A)$ to be $A$. For $k$ nonnegative and even we define $F^k_\mathcal{P}(A)$ to be the greatest set $B$ such that $B = T^k_\mathcal{P}(F^{k-1}_\mathcal{P}(A[k := B]))$. In fixed point notation this can be written as

$$F^k_\mathcal{P}(A) = \nu B.\ T^k_\mathcal{P}(F^{k-1}_\mathcal{P}(A[k := B]))$$

For nonnegative odd values of $k$ the definition of $F^k_\mathcal{P}$ is analogous but using a least fixed point ($\mu$) instead of a least fixed point ($\nu$). The data base $F^k_\mathcal{P}(A)$ is the database in which the meaning of predicates of priority greater than $k$ is taken from A while the meaning of predicates of priority no greater than $k$ is computed from their definition. Note that predicates of even priority are given greatest fixed point meanings while predicates of odd priority are given least fixed point meanings. We will now prove the following theorem.

**Theorem 1**
*$[\![\mathcal{P}]\!] = F^n_{\mathcal{P}}(\emptyset)$ where $n$ is the largest priority of any predicate in $\mathcal{P}$.*

The proof is similar to the proof of Theorem 6.1 in [**?**] (or Theorem 3.2 in [**?**]). The proof is by induction on the priority $k$ in the operators $F^k_{\mathcal{P}}$. More specifically, for any program $\mathcal{P}$, integer $k$, and database $A$ we define a $k$-$\mathcal{P}$-$A$-derivation to be a tree $r$ where each node $w$ is labeled with a ground atom $r(w)$ and has children $C(w)$ and where for each node $w$ we either have that the priority of $r(w)$ is greater than $k$ and $r(w) \in A$ or the priority of $r(w)$ is no greater than $k$ and $r(w) \leftarrow r(C(w))$ is a ground instance of a clause in $\mathcal{P}$. A node in a $k$-$A$-$\mathcal{P}$-derivation which not justified by a clause in $\mathcal{P}$ will be called a premise of that derivation. A $k$-$\mathcal{P}$-$A$-derivation tree is said to satisfy parity if for every infinite path in that tree the largest priority occurring infinitely often on that path is even. We define $[\![\mathcal{P}]\!]_{k,A}$ to be the set of ground atoms $\Phi$ such that there exists a parity-satsfying $k$-$A$-$\mathcal{P}$-derivation of $\Phi$. One can show by induction on $k$ that for any database $A$ we have $F^k_{\mathcal{P}}(A) = [\![\mathcal{P}]\!]_{k,A}$. Theorem 1 then follows by taking $k = n$. The full proof is given in appendix A.

# 3 Memorylessness and Determinancy

In this section we give two fundamental facts about afp logic programs — memorylessness and determinancy. Both of these follow from well known results for parity games. However, we give brief proof sketches here for the sake of completeness. Memorylessness is used in section **??** but determinancy is not used in the remainder of the paper.

First we consider memorylessness. Let $\mathcal{P}$ be any afp logic program. We define a choice function for $\mathcal{P}$ to be a function $c$ mapping $[\![\mathcal{P}]\!]$ to finite sets of of atoms such that for any ground atom $\Phi \in [\![\mathcal{P}]\!]$ we have that $\Phi \leftarrow c(\Phi)$ is an instance of a clause in $\mathcal{P}$. For any given choice function $c$ for $\mathcal{P}$ it is easy to see that for any $\Phi \in [\![\mathcal{P}]\!]$ there is a unique $\mathcal{P}$-derivation $r$ of $\Phi$ such that at every node $p$ in $r$ we have $r(C(p)) = c(r(p))$. We will call this the $c$-derivation of $\Phi$. A choice function $c$ will be said to satisfy parity if for every $\Phi \in [\![\mathcal{P}]\!]$ we have that the $c$-derivation of $\Phi$ satisfies parity. It is obvious that for any program $\mathcal{P}$ there exists choice functions for $\mathcal{P}$ but it is not obvious that there exists a choice function satisfying parity. The following theorem states the memorylessness property of afp logic programs.

**Theorem 2**
*There exists a parity-satsifying choice function for every afp logic program.*

By theorem 1 it suffices to show that for any data base $A$ and integer $k$ there exists a choice function defined on the set $F^k_{\mathcal{P}}(A)$ which yields a parity-satsifying $k$-$A$-$\mathcal{P}$-derivation. The proof is by induction on $k$ and is essentially identicle to the proof in appendix A that $F^k_{\mathcal{P}}(A) \subseteq [\![\mathcal{P}]\!]_{k,A}$, i.e., that every atom in $F^k_{\mathcal{P}}(A)$ has a $k$-$A$-$\mathcal{P}$-derivation.

We now state the determinancy property and give a brief proof sketch. We define a $\mathcal{P}$-coderivation to be a (possibly infinite) tree $r$ in which every node is labeled with a negated ground atom, i.e., a literal of the form $\neg\Phi$, such that for each position $p$ in $r$ and each instance $\Phi \leftarrow B$ of a clause in $\mathcal{P}$ such that $r(p) = \neg\Phi$ we have that $r(C(p))$ contains a negated atom $\neg\Psi$ with $\Psi \in B$. A coderivation is said to satisfy parity if for every infinite path in the coderivation the maximum priority occuring infinitely often is odd. It is useful to consider the least fixed point case — suppose that $\mathcal{P}$ is a least fixed point program. A $\mathcal{P}$-coderivation of $\neg\Phi$ provides a way of defeating any attempt to use a clause from $\mathcal{P}$ to derive $\Phi$ — for every instance $\Phi \leftarrow B$ of a clause in $\mathcal{P}$ the coderivation provides a way of defeating some atom in the body $B$. If there is a $\mathcal{P}$-coderivation of $\neg\Phi$ then there can not be any finite $\mathcal{P}$-derivation of $\Phi$. More generally, for any afp program $\mathcal{P}$ it is possible to show that if there is a parity-satisfying $\mathcal{P}$-coderivation of $\neg\Phi$ then $\Phi \notin [\![\mathcal{P}]\!]$. To see this suppose there is parity-satisfying $\mathcal{P}$-coderivation of $\neg\Phi$ and consider any $\mathcal{P}$-derivation of $\Phi$. The $\mathcal{P}$-coderivation can be used to select a "defeatable" child of the root of the $\mathcal{P}$-derivation. We can then select a defeatable child of that child and so on. For every node encountered in this way the definition of a $\mathcal{P}$-coderivation ensures that the process can be continued so this leads to an infinite path in both the $\mathcal{P}$-derivation and the $\mathcal{P}$-coderivation. The maximum priority occuring infinitely often on this path must be odd and hence the given $\mathcal{P}$-derivation does not satisfy parity. So if there is a parity-satisfying $\mathcal{P}$-coderivation of $\neg\Phi$ then there is no parity-satisying $\mathcal{P}$-derivation of $\Phi$. The determinancy property states the converse.

**Theorem 3**
*If $\Phi \notin [\![\mathcal{P}]\!]$ then there exists a $\mathcal{P}$-coderivation of $\neg\Phi$.*

This theorem follows from determinancy for Borel games. However, it is possible to sketch a simple direct proof based on the fixed point characterization of $[\![\mathcal{P}]\!]$. First we define the game-theoretic dual of the operator $T^k_{\mathcal{P}}$. More specifically, we define the operator $\overline{T}^k_{\mathcal{P}}$ so that $\overline{T}^k_{\mathcal{P}}(A)$ is the set of literals $\neg\Phi$ such that for each instance of a clause in $\mathcal{P}$ of the form $\Phi \leftarrow B$ there is some $\Psi \in B$ such that $\neg\Psi \in A$. Now we define the operators $\overline{F}^k_{\mathcal{P}}$ as the dual of $F^k_{\mathcal{P}}$. More specifically, for $k$ negative we $\overline{F}^k_{\mathcal{P}}(A) = A$. For $k$ non-negative and even we define $\overline{F}^k_{\mathcal{P}}(A)$ as follows.

$$\overline{F}^k_{\mathcal{P}}(A) = \mu B.\ T^k_{\mathcal{P}}(\overline{F}^{k-1}_{\mathcal{P}}(A[k := B]))$$

For $k$ non-negative and odd the definition is identicle but using a greatest fixed point rather than a least fixed point. Now for any database $A$ define $\neg A$ to be the set of literals of the form $\neg\Psi$ for $\Psi \notin A$. Basic properties of least fixed point expressions can be used to prove, by induction on $k$, that $\overline{F}^k_{\mathcal{P}}(\neg A) = \neg F^k_{\mathcal{P}}(A)$. In particular, by theorem 1 we have that $\neg[\![\mathcal{P}]\!] = \overline{F}^n_{\mathcal{P}}(\neg\emptyset)$ where $n$ is the largest priority in $\mathcal{P}$. If $n$ is the largest priority in $\mathcal{P}$ then $\overline{F}^n_{\mathcal{P}}(A) = \overline{F}^n_{\mathcal{P}}(\emptyset)$ for any set $A$. So we have $\neg[\![\mathcal{P}]\!] = \overline{F}^k_{\mathcal{P}}(\emptyset)$. It now suffices to prove that $\overline{F}^k_{\mathcal{P}}(\emptyset)$ equals the set of negative literals for which there exists a parity-satisfying $\mathcal{P}$-coderivation. The proof is directly analogous to the proof of theorem 1.

# 4 Program Relaxations

Here we give an example of a method of inferring types for a logic program by relaxing the program to a uniform program. Let $[\![\mathcal{P},\ p]\!]$ denote the set of terms $t$ such that $p(t) \in [\![\mathcal{P}]\!]$. We will call a program $Q$ a relaxation of a program $\mathcal{P}$ if $[\![\mathcal{P},\ p]\!] \subseteq [\![Q,\ p]\!]$ for any predicate symbol $p$ occurring in $\mathcal{P}$. If we want to prove that some set $[\![\mathcal{P},\ p]\!]$ is empty, it suffices to construct a relaxation $Q$ of $\mathcal{P}$ and show that $[\![Q,\ p]\!]$ is empty. To show that some set $[\![\mathcal{P},\ p]\!]$ contains all terms it suffices to show that the complement is empty. Most liveness and safety properties are easily complemented although the general question of whether the full class of sets expressible by afp programs is closed under complementation remains open.

For any afp program $\mathcal{P}$ we now define a relaxation $\mathcal{P}^{\#}$ corresponding to the Heintze and Jaffar set based analysis of logic programs [**?**]. For each clause $H[x_1,\ \ldots,\ x_n] \leftarrow B[x_1,\ \ldots,\ x_n]$ containing variables $x_1, \ldots, x_n$ the following clauses are included in $\mathcal{P}^{\#}$.

$$\tau_1(x_1) \leftarrow B[x_1,\ \ldots,\ x_n]$$
$$\vdots$$
$$\tau_n(x_n) \leftarrow B[x_1,\ \ldots,\ x_n]$$
$$H[y_1,\ \ldots,\ y_k] \leftarrow \tau_{h_1}(y_1),\ \ldots,\ \tau_{h_k}(y_k)$$

Here $\tau_i$ is a fresh predicate symbol of priority 0 representing the "type" of the variable $x_i$. The atom $H[y_1,\ \ldots,\ y_k]$ is the result of replacing each occurrence of a variable $x_i$ in $H$ by a fresh variable $y_j$ so that $H[y_1,\ \ldots,\ y_k]$ is linear, i.e., no variable occurs more than once. The predicate $\tau_{h_j}$ in the body of the last clause equals $\tau_i$ where $x_i$ is the variable replaced by $y_j$. For example, if $\mathcal{P}$ contains the clause $P(x,\ x,\ y) \leftarrow Q(x,\ y)$ then $\mathcal{P}^{\#}$ contains the clauses $\tau_1(x) \leftarrow P(x,\ y)$, $\tau_2(y) \leftarrow Q(x,\ y)$ and $P(x_1,\ x_2,\ y) \leftarrow \tau_1(x_1),\ \tau_1(x_2),\ \tau_2(y)$. It is easy to see that any parity-satisfying $\mathcal{P}$-derivation can be converted to a parity-satisfying $\mathcal{P}^{\#}$-derivation. Hence $\mathcal{P}^{\#}$ is a relaxation of $\mathcal{P}$.

The relaxation $\mathcal{P}^{\#}$ can be converted in linear time into a uniform program $Q$ such that for any of the type predicates $\tau_i$ used in $\mathcal{P}^{\#}$ we have that $[\![\mathcal{P}^{\#}, \tau_i]\!]$ equals $[\![Q, \tau_i]\!]$. Furthermore, the uniform program $Q$ has the property that for any predicate $p$ used in $\mathcal{P}$ there is a monadic predicate $q$ used in $Q$ such that $[\![\mathcal{P}^{\#},\ p]\!]$ is empty if and only if $[\![Q,\ q]\!]$ is empty.

## 5 Regularity of Uniform Programs

We now prove the first main theorrem of this paper.

**Theorem 4 (Regularity)**
*If $P$ is a uniform afp program then the set $[\![P,\ p]\!]$ is regular, i.e., definable by a regular afp program.*

The proof gives a *non-constructive* method of converting a uniform program to an equivalent alternating-regular program. This suffices since it is well known that the sets definable by alternating-regular programs are regular. The fundamental intuition is that projection clauses can be "short-circuited" by alternation clauses. For example, consider the projection clause $p(x) \leftarrow q(f(x,\ g(x,\ y)))$. Suppose the program also contains the reduction clauses $r(g(x,\ y)) \leftarrow h(x),\ w(y)$ and $q(f(x,\ u)) \leftarrow s(x),\ r(u)$. If one can determine that there exists a term $t$ such that $w(t) \in [\![P]\!]$ then from these clauses we can infer the short-circuit alternation clause $p(x) \leftarrow s(x),\ h(x)$. Given enough of these alternation clauses it is possible to enumerate all the ways of short circuiting the projection clauses and the projection clauses become redundant.

In the case of purely least fixed point programs or purely greatest fixed point programs this basic intuition can be used as the foundation for a constructive algorithm for converting uniform programs to alternating automata. In the case of afp programs things are more complex. One problem is that the meaning of afp programs is not invariant under the addition of derived clauses. If the program contains $p(x) \leftarrow q(x)$ and $q(x) \leftarrow r(x)$ the introduction of the derived clause $p(x) \leftarrow r(x)$ may allow derivation trees which satisfy the parity condition by virtue of the fact that uses of the predicate $q$ have been eliminated from certain derivation paths. It is possible, however, to formulate a nonstandard notion of derived clause such that the program remains invariant under the addition of derived clauses.

To formulate a notion of derived clause that preserves program meaning we first introduce additional predicate symbols. For each predicate $p$ occurring in $\mathcal{P}$ and each priority $k$ at least as large as the priority of $p$ we introduce a new predicate $p_k$ with priority $k$ and the clause $p_k(x) \leftarrow p(x)$. For any ground atom $\Phi$ involving a predicate in $\mathcal{P}$ we define $\Phi_k$ to be the result of replacing $p$ by $p_k$. We define an open $\mathcal{P}$-derivation to be a (possibly infinite) derivation tree with a specified (possibly infinite) set of non-root leaf positions called assumptions such that at every non-assumption position $p$ in $r$ we have that $r(p) \leftarrow r(C(p))$ is an instance of a clause in $\mathcal{P}$. We say that a clause $H \leftarrow B$ is derivable from an arbitrary afp program $\mathcal{P}$ if there exists a ground substitution $\rho$ mapping the variables in $H \leftarrow B$ to fresh constants and an open $\mathcal{P}$-derivation $r$ such that the root of $r$ is labeled with $\rho(H)$ and every premise of $r$ is of the form $\rho(\Psi)$ where $B$ contains $\Psi_k$ where $k$ is the largest priority on the path from the root to the assumption.

**Derived Rule Soundness Lemma**
*For any afp logic program $\mathcal{P}$, if $\mathcal{P}'$ consists of $\mathcal{P}$ plus rules of the form $p_k(x) \leftarrow p(x)$ plus rules derivable from $\mathcal{P}$ then $[\![\mathcal{P}', p]\!] = [\![\mathcal{P}, p]\!]$ for any predicate $p$ in $\mathcal{P}$.*

To prove the derived rule soundness lemma it suffices to show that any parity-satisfying $\mathcal{P}'$-derivation can be converted to a parity-satisfying $\mathcal{P}$-derivation. Let $r$ be any parity-satisfynig $\mathcal{P}'$-derivation. Starting at the root of the derivation we repeatedly replace each use of a derived rule by an appropriate renaming of an open $\mathcal{P}$-derivation justifying that rule. Since the assumptions in open $\mathcal{P}$-derivations are at non-root positions this process must push the uses of derived rules strictly lower in the derivation. Hence this infinite process yields an infinite $\mathcal{P}$-derivation $r'$. To prove that this infinite $\mathcal{P}$-derivation satisfies parity consider any infinite path in $r'$. This infinite path is composed of segements from $r$ and segments from the introduced open derivations. Each segment inside an introduced open derivation corresponds from a three node path in $r$ from the head of a derived rule to a predicate of the form $p_k$ and then the predicate $p$ where $k$ is the maximum priority on the corresponding of the path in $r'$. Hence the maximum priority occuring infinitely on the path in $r'$ equals the maximum priority occuring infinitely on the corresponding path in $r$. Since the maximum infinite priority in $r$ path must be even so must the maximum infinite priority in the path in $r$.

We now prove theroem 4. Let $\mathcal{P}$ be any uniform afp logic program. Now let $\mathcal{P}'$ consist of all the reduction and alternation clauses in $\mathcal{P}$, plus all alternation clauses derivable from $\mathcal{P}$, plus all clauses of the form $q_k(x) \leftarrow q(x)$. By the soundness lemma for derived clauses we have that $[\![\mathcal{P}', p]\!] \subseteq [\![\mathcal{P}]\!]$ for any $p$ in $\mathcal{P}$. We now show that $[\![\mathcal{P}, p]\!] \subseteq [\![\mathcal{P}', p]\!]$. Consider a $\mathcal{P}$-derivation $r$ using a projection rule $p(x) \leftarrow q(w)$ under substitution $\rho$ at the root position. For any variable $y$ in the projection clause let $C_y(y)$ be the set of atoms $h_k(y)$ such that there is a non-root node in $r$ labeled with $h(\rho(y))$ and such that the maximum priority on the path from the root to that node is $k$. Now consider the following alternation clause where $x, y_1, \ldots, y_n$ are all the variables of the projection clause $p(x) \leftarrow q(w)$.

$$p(x) \leftarrow C_x(x), C_{y_1}(y_1),\ \ldots\ ,\ C_{y_n}(y_n)$$

by definition, this clause is derivable from $\mathcal{P}$.[3] So any use of a projection clause can be replaced by a use of a derived alternation clause. Given an arbitrary $\mathcal{P}$-derivation we can repeatedly replace a shallowest use of a projection clause by a derived alternation clause. This infinite process yields in a $\mathcal{P}'$-derivation. An argument similar to that used in the derived rule soundness lemma shows that this derivation must satisy parity.

Although we have not shown how to construct the derivable alternation clauses, we can bound the size of the resulting alternating automaton $\mathcal{P}'$. If $m$ is a number of predicate letters used in $\Psi$ and $n$ the maximum priority of these predicates, then $\Psi'$ uses at most $mn$ new predicates, but perhaps $2^{O(mn)}$ new clauses. Note that we can further transform $\Psi'$ into a *nondeterministic* automaton by the construction of [**?**] that depends exponentially only on the number of *states* (that is, predicate letters in our setting). Thus we can obtain a nondeterministic automaton equivalent to the program $\Psi$ of the size $2^{|\mathcal{P}|^{0(1)}}$.

Given a method of testing emptyness for uniform afp programs it is possible to make the construction given here effective and actually compute a regular afp program equivalent to the given program $\mathcal{P}$ although we do not explicitly give such a construction here. A method of testing emptyness for uniform programs is given in section 7.

# 6 Parity Subject Reduction

The correctness proof for the method of testing emptyness for uniform afp programs requires proving a form of equivalences between a given uniform afp program and a certain alternating-regular program. Consider the problem of proving that two given afp programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent. This can be done by showing that any parity-satisfying $\mathcal{P}$-derivation can be translated into a parity-satisfying $\mathcal{P}'$-derivation and vice-versa. Such translation can often be done by a process of local rewriting — each use of an inference rule in $\mathcal{P}'$ is translated to a corresponding use of an inference rule in $\mathcal{P}$. However, for afp logic programs derivations are generally infinite and the translation can only be completed in the limit of an infinite rewriting process. In some cases it can be difficult to prove that transfinite rewriting preserves parity, i.e., if we start with a parity-satisfying derivation then the limit of infinite rewriting also satisfies parity. In this section we prove a parity subject reduction theorem stating that under a fairly general notion of reduction, parity is preserved by transfinite reduction. For this section we need a more rigorous treatment of trees.

We formally define a position to be a finite sequence of natural numbers. For example, the sequence $\langle 2, 4, 1\rangle$ names the position reached by taking the second child of the root, the fourth child of that, and then the first child of that. We say that position $p$ is below position $q$ if the sequence $q$ is a prefix of the sequence $p$. We say that $p$ is above $q$ if $q$ is below $p$. We say that $p$ is to the left of $q$ if neither is a prefix of the other and at the first place where they differ the number in $p$ is less than the number in $q$. We say that $p$ is to the right of $q$ if $q$ is to the left of $p$. Note that the relations "below or left" and "above or right" are both total orders on positions (corresponding to the order of words in a dictionary). We write $p.m$ for the position that results from adding $m$ to the end of the sequence $p$. A position of the form $p.m$ is called a child of $p$ and $p$ is called the parent of $p.m$. Two positions of the form $p.m$ and $p.n$ for $m \neq n$ are called siblings. We also write $p \vee q$ for the deepest position above both $p$ and $q$.

Let $L$ be any set of "labels" such that each label is assigned a natural number called a priority. An $L$-tree is a function from positions to $L \cup \{\perp\}$ where $\perp$ is a distinguished value not in $L$ and where $r$ must be such that if $r(p) = \perp$ then $r(q) = \perp$ for any child or right-sibling $q$ of $p$. Note that we allow trees to be infinitely branching. We say that position $p$ is "in" $r$ if $r(p) \neq \perp$. We say that $p$ is below $q$ in $r$ if both $p$ and $q$ are in $r$ and $p$ is below $q$. A similar statement holds for $p$ above $q$ in $r$, $p$ a

[3]This construction can be applied to clauses other than projection clauses. However, in the general case the resulting derived clause has a nontrivial head and therefore is not an alternation clause.

child of $q$ in $r$, and $p$ a sibling of $q$ in $r$. We write $\Omega_r(p)$ for the priority of the label $r(p)$. If $q$ is below $p$ in $r$ we write $\Omega_r(p,\ q)$ for the maximum priority occuring on the path from $p$ to $q$ including positions $p$ and $q$. A infinite sequence of $L$-trees $r_1, r_2, r_3, \ldots$ is said to converge to $r$ if for every position $p$ there exists a $k$ such for all $j > k$ we have $r_j(p) = r(p)$.

In presenting the general conditions under which infinte rewriting preserves parity it will be useful to have a simple example in mind. We define the set of let-terms by the following grammar.

$$t ::= x \mid f(t_1,\ \ldots,\ t_n) \mid \mathrm{let}_i(b_j(x,\ t_1),\ t_2)$$

We will be interested in rewriting steps that replace $\mathrm{let}_i(b_j(x,\ s),\ t)$ by $t[x := s]$ (with renaming of bound variables so as to avoid the capture of free variables in $s$). This is simply the classical substitution of a value for a variable in let expressions as occur in programming languages such as ML. The significance of subscript $i$ on $\mathrm{let}_i$ and of the "function" $b_i$ involves priorities and is discussed below.

We will be concerned with priorities and the parity condition. We assign each function symbol $f$ appearing in let terms a priority. For each natural number $i$ there is a version of the let symbol, $\mathrm{let}_i$, which has priority $i$. For each natural number $j$ there is a symbol $b_j$ of priority $j$ that can be used in "binding pairs" in let expression. We also assume that each variable is assigned a priority and that there are are uncountably many variables of each priority (to allow renaming in countably infinite terms). A let term is an $L$-tree where the labels are the symbols of the form $f$, $x$, $\mathrm{let}_i$ and $b_j$. A let term will be called well-formed if it is a (possibly infinite) term generated by the above grammar and for every subterm of the form $\mathrm{let}_i(b_j(x,\ s),\ t)$ the priority of $x$ is the same as the priority of the top level symbol in $s$; the priority of $\mathrm{let}_i$ is the same as the priority of the top level symbol in $t$; and the maximum priority on any path from $t$ to a free occurence of $x$ in $t$ equals the priority of $b_j$. Note that this ensures that the path from the root to $s$ in $\mathrm{let}_i(b_j(x,\ s),\ t)$ has the same maximum priority as any path from $t$ to a free occurence of $x$ in $t$. A term satsifies the parity condition if for every infinite path in that path the maximum priority occuring infinitely often on that path is even. The parity subject reduction theorem given below implies that if $t$ is a well-formed let term satisfying the parity condition then any term that can be derived as the limit of an infinite sequence of reductions from $\mathrm{let}_i(b_j(x,\ s),\ t)$ to $t[x := s]$ also satisfies the parity condition.

Consider a let expression $\mathrm{let}_i(b_j(x,\ s),\ t)$. Certain positions in this expression can be viewed as "standing for" other positions. Most notably, any free occurance of $x$ in $t$ stands for $s$. We can define a function, which we call a value function, that maps positions to the positions they stand for. We let $V(p)$ be the position that $p$ stands for which we will also call the value of $p$. If $p$ is a free occurence of $x$ in $t$ then $V(p)$ is the occurence of $s$ in $\mathrm{let}_i(b_j(x,\ s),\ t)$. If $p$ is the root position (the position of the let expression itself) then $V(p)$ is the position of $t$. If $p$ is not assigned a position by these rules then $V(p) = p$. Intuitively, $V(p)$ is the position that $p$ "evaluates to". We leave it to the reader to verify that these conventions for assigning values to positions in let terms defines a continuus value scheme for well-formed let terms as defined below.

**Value Schemes** Let $r$ be an $L$-tree and let $V$ be a function from positions in $r$ to positins in $r$. We call $V$ below-or-left if $V(p)$ is either $p$ or is below or left of $p$ for all positions $p$ in $r$. Any position of the form $V(p) \vee p$ where $V(p) \neq p$ will be called a binding position. We say that $V$ is locally-binding provided that for any position $p$ there are no binding positions between $V(p) \vee p$ and $V(p)$, i.e., below $V(p) \vee p$ but above $V(p)$. We say that $V$ is priority-preserving for any position $p$ we have $\Omega_r(V(p)) = \Omega_r(p)$ and $\Omega_r(V(p) \vee p,\ V(p)) = \Omega_r(V(p) \vee p,\ p)$. We say that $V$ is a value function if it is below-or-left, locally-binding, and priority-preserving. A value scheme is a function $V$ defined on a class of $L$-tree, called the $V$-well-formed trees, such that for each tree $r$ in the domain of the scheme $V$ we have that $V_r$ (the value of $V$ on $r$) is a value function on $r$. The scheme $V$ is called continuous if for for any infinite sequence of $V$-well-formed trees $r_1$, $r_2$, $r_3$, $\ldots$ which converges to $r$ we have that $r$ is $V$-well-formed and for any position $p$ in $r$ the sequence $V_{r_i}(p)$ converges, i.e., there exists a $k$ such that for all $j > k$ we have that $p$ is in $r_j$ and $V_{r_j}(p) = V_r(p)$.

The rewriting step from $C[\mathrm{let}(b_j(x,\ s),\ t)]$ to $C[t[x := s]]$ can be viewed as replacing the subtree tree at a given position by the tree at the value of that position. There is a natural mapping from positions in the term $C[t[x := s]]$ to positions in the term $C[\text{let}(b_j(x,\ s),\ t)]$. We leave it to the reader to verify that this natural mapping is a reduction map in the sense defined below and therefore that the reduction from $C[\text{let}(b_j(x,\ s),\ t)]$ to $C[t[x := s]]$ is a $V$-reduction as defined below.

**$V$-reducibility** For any position $p$ in a term $r$ and value map $V$ on $r$ we define $V^k(p)$ by $V^0(p) = p$ and $V^{k+1}(p) = V(V^k(p))$. We define $V^*(p)$ to be the set of positions of the form $V^k(p)$ for $k \geq 0$. Now let $V$ be a value scheme and let $r$ and $s$ be two $V$-well-formed terms. We say that a function $f$ from the positions in $s$ to the positions in $r$ is a reduction map if for every position $p$ in $s$ and for every child $p.m$ of $p$ in $s$ we have that $f(p).m$ is in $r$ and the following conditions hold.

$$\Omega_r(f(p)) = \Omega_s(p)$$

$$f(V_s(p)) \in V_r^*(f(p))$$

$$f(p.m) \in V_r^*(f(p).m)$$

If there exists a reduction map from $s$ to $r$ under value scheme $V$ then we say that $r$ is $V$-reducible to $s$.

Appendix B gives proofs of the following theorems.

**Transitivity of $V$-redicibility** For any value scheme $V$ we have that $V$-reducibility is transitive.

**Continuity of $V$-reducibility** We say that $r$ $V$-reduces to $s$ at depth $d$, written $r \xrightarrow{d} s$, if there exists a reduction map $f$ from $s$ to $r$ such that for all positions $p$ of depth less than $d$ we have that $r(p) = s(p)$ and $f(p) = p$. We define an ever-deeper reduction sequence to be an infinite reduction chain $r_1 \xrightarrow{d_1} r_2 \xrightarrow{d_2} r_3 \xrightarrow{d_3} \ldots$ such that for all $k > 0$ then there exists an $i$ such that $d_j > k$ for all $j > i$. If $V$ is continuous and $r$ is the limit of an every-deeper reduction chain $r_1, r_2, r_3, \ldots$ then $r_1$ $V$-reduces to the limit $r$.

**Theorem 5 (Parity Subject Reduction)**
*If $V$ is a value scheme on $L$-trees, and $r$ is a $V$-well-formed tree, we have that if $r$ satisfies parity and $V$-reduces to $s$ then $s$ satisfies parity.*

# 7 The Main Theorem

**Theorem 6**
*If $\mathcal{P}$ is uniform then the emptiness of $[\![\mathcal{P}, p]\!]$ can be determined in $2^{|\mathcal{P}|^{O(1)}}$ time.*

The theorem is proved by giving a "sound" and "complete" translation of the given uniform program $\mathcal{P}$ into an alternating-regular program $\mathcal{P}'$. The notions of soundness and completeness are clearest in the case of least fixed point programs and we first present a proof for this special case. The least fixed point proof is then generalized to handle greatest fixed points and finally afp programs.

It is not difficult to show that any uniform afp program is equivalent to one consisting of only regular and projection clauses. In the remainder of this section we assume that $\mathcal{P}$ contains only regular and projection clauses.

## 7.1 The lfp Case

As a simple first case we consider least fixed point (lfp) programs. Although our semantics of lfp programs allows infinite terms, any lfp derivation must be finite and hence can only "look at" a finite portion of a term. Hence if $p(t) \in [\![\mathcal{P}]\!]_\mu$ where $t$ is infinite then there exists a fintie term $t'$ which results by replacing certain subterms with a fresh constant $\perp$ such that $t' \in [\![\mathcal{P}]\!]$. So to determine emptyness in the lfp case we can assume without loss of generality that all terms are finite.

It is known that if $\mathcal{P}$ is uniform then its least fixed point meaning over finite terms is a regular tree language (of finite trees) and that the emptyness of $[\![\mathcal{P}, p]\!]$ is decidable in single exponential time in the size of $\mathcal{P}$ [**?**]. However, we do not know of any way of generalizing the standard proof to the general afp case. In this section we describe a proof for the lfp case which does generalize.

Given a uniform least fixed point program $\mathcal{P}$ we construct an alternating-regular program $\mathcal{P}'$ that runs on "annotated terms". To define the annotated terms we introduce a constant symbol $c_p$ for each predicate $p$ occuring in $\mathcal{P}$, a constant symbol $c_x$ for each variable $x$ occurring in $\mathcal{P}$, and a function symbol $\Pi_x$ of two arguments for each variable $x$ occuring in $\mathcal{P}$. Finite annotated terms are defined by the following grammar where $\perp$ is a special new constant symbol; $c_x$ and $c_p$ ranges over the introduced constants; $\tau$ ranges over "types", i.e., sets of constants of the form $c_p$; $f$ ranges over function symbols occurring in $\mathcal{P}$; $t$ ranges over annotated terms; and $\sigma$ ranges over substitutions, i.e., a set of pairs of a constant of the form $c_x$ and a value.

$$\tau ::= emptytype \mid type(c_p,\ \tau)$$
$$t ::= \bot \mid var(c_x,\ \tau) \mid f(t_1,\ \ldots,\ t_n) \mid \Pi_x(\sigma,\ t)$$
$$\sigma ::= emptysubst \mid subst(c_x,\ t,\ \sigma)$$

For any substitution $\sigma$ as defined by the above grammar and variable $x$ occurring in $\mathcal{P}$ we define $\sigma(x)$ to be the first term associated with $c_x$ in $\sigma$ or the special term $\bot$ if $\sigma$ does not associate $c_x$ with any term. A term of the form $var(c_x,\ \tau)$ represents a typed variable where $c_x$ specifies the name of the variable and $\tau$ specifies the "type", i.e., a set of predicate symbols. We say that an occurence of $var(c_x,\ \tau)$ in $t$ is bound if it occurs inside the second argument in an expression of the form $\Pi_y(\sigma,\ s)$. An occurance that is not bound is called free. An annotated term with no free occurrences of typed variables wi called closed.

We construct a alternating-regular lfp program $\mathcal{P}'$ such that there exists a finite closed annotated term $t$ in $[\![\mathcal{P}', p]\!]$ if and only if $[\![\mathcal{P}, p]\!]$ contains a finite term, and hence if and only if $[\![\mathcal{P}, p]\!]$ ofer infinite terms is nonepmty. It turns out that this is sufficient to prove the desired result. More specifically, the property of being a finite closed annotated term as defined by the above grammar can be defined by a regular lfp program. Given any program $\mathcal{P}'$ satisfying the above condition we ca an coinjoin $\mathcal{P}'$ with a program defining finite closed annotated terms to give an alternating-regular program $\mathcal{P}''$ such that $[\![\mathcal{P}'', p]\!]$ is nonempty if and only if $[\![\mathcal{P}, p]\!]$ is nonempty.

Before presenting the clauses of $\mathcal{P}'$ we define a semantics for the atoms used in the $\mathcal{P}'$ program. The first step is to define a semantics for annotated terms. For any term $t$ and substitution $\sigma$ (as defined by the above grammar) we define $\sigma(t)$ to be the result of replacing each free occurance of a typed variable $var(c_x,\ \tau)$ in $t$ by $\sigma(x)$. Intuitively, the term $\Pi_x(\sigma,\ t)$ represents the pair of terms $\sigma(x)$ and $\sigma(t)$. The term $\sigma(t)$ is a "context" that typically contains the term $\sigma(x)$. So terms defined by the above grammar can be viewed as ordinary terms (with typed variables) annotated with containing contexts. More formally, we define the following deletion operator for removing annotations.

$$D(\Pi_x(\sigma,\ t)) = D(\sigma(x))$$
$$D(f(t_1,\ \ldots,\ t_n)) = f(D(t_1),\ \ldots,\ D(t_n))$$
$$D(var(c_x,\ \tau)) = var(c_x,\ \tau)$$

Note that the deletion operator $D$ removes all occurrences of the function symbols $\Pi_x$ but preserves typed variables.

We now address the semantics of the atoms used in the $\mathcal{P}'$. If $t$ is a closed annotated term then $p(t)$ is true if $p(D(t)) \in [\![\mathcal{P}]\!]$. Note however that in general $D(t)$ may contain typed variables. To handle typed variables in $D(t)$ we introduce a set of regular clauses $Q$ such that $p(var(c_x,\ \tau))$ is provable using clauses in $Q$ if and only if $c_p$ is in the type $\tau$. The clause set $Q$ is given in appendix C. If $t$ is an abtriary annotated term (open or closed) then we say that $p(t)$ is $\mathcal{P}$-true if there is a finite $\mathcal{P} \cup Q$-derivation of $p(D(t))$ such that the only typed variables occuring in that derivation are those that occur free in $t$.

In addition to the predicates used in $\mathcal{P}$, the program $\mathcal{P}'$ uses certain additional auxilliary predicates. For each predicate $p$ and variable $x$ in $\mathcal{P}$ there is a predicate $pr_{p,x}$ where $pr_{p,x}(\sigma)$ states that the substitution $\sigma$ "provides" the assertion $p(x)$. More rigorously, we define $pr_{p,x}(\sigma)$ to be $\mathcal{P}$-true if $\sigma(x)$ is such that $p(\sigma(x))$ is $\mathcal{P}$-true. For each predicate $p$ and variable $x$ in $\mathcal{P}$ there is a predicate $na_{p,x}$ where $na_{p,x}(t)$ states that $t$ does not assume $p(x)$. More rigorously, we define $na_{p,x}(t)$ to be $\mathcal{P}$-true provided that $t$ does not contain any free occurrence of a typed variable of the form $var(c_x,\ \tau)$ where $\tau$ contains $p$. For each predicate $p$ and variable $x$ in $\mathcal{P}$ there is an assertion $ok_{p,x}$ where $ok_{p,x}(\Pi_x(\sigma,\ t))$ is defined to be $\mathcal{P}$-true if either $na_{p,x}(t)$ is $\mathcal{P}$-true or $pr_{p,x}(\sigma)$ is $\mathcal{P}$-true. There is also a predicate $ok$ where $ok(\Pi_x(\sigma,\ t))$ is $\mathcal{P}$-true if for each predicte $p$ and variable $x$ in $\mathcal{P}$ we have that $ok_{p,x}(\Pi_x(\sigma,\ t))$ is $\mathcal{P}$-true, i.e., for each predicate $p$ and variable $x$ we have that if $t$ assumes $p(x)$ then $\sigma$ provides $p(x)$. Finally, we define a $\mathcal{P}$-pattern to be a term appearing as a subterm of the left hand side of a projection clause in $\mathcal{P}$. For each $\mathcal{P}$-pattern $w$ there is a predicate $=_w$. To define the sematnics of $=_w$ we first define the operation $D'$ on annotated terms where $D'(t)$ is the result of replacing each typed variable $var(c_y,\ \tau)$ in $D(t)$ by the variable $y$. The assertion $=_w(t)$ is defined to be $\mathcal{P}$-true if $D'(t) = w$, i.e., $t$ is a typed variant of the pattern $w$.

The complete program $\mathcal{P}'$ for the lfp case is given in appendix C. Each clause in $\mathcal{P}'$ is sound relative to the semantics defined above, i.e., if the antecedents are $\mathcal{P}$-true then the conclusion is $\mathcal{P}$-true. There are three particularly significant types of clauses in $\mathcal{P}'$. First, each regular clause in $\mathcal{P}$ is included in $\mathcal{P}'$. Second, for each projection clause $p(x) \leftarrow q(w)$ in $\mathcal{P}$ we include the following in $\mathcal{P}'$ where $\sigma$ and $t$ are variables of the clause.

$$p(\Pi_x(\sigma,\ t)) \leftarrow\ ok(\Pi_x(\sigma,\ t)),\ =_w(t),\ q(t) \quad (4)$$

This clause is easily translated into regular and alternation clauses. Third, we have "pass-the-buck" clauses of the following form where again $\sigma$ and $t$ are variables of the clause.

$$p(\Pi_x(\sigma,\ t)) \leftarrow pr_{p,x}(\sigma)$$

Soundness is trivial for all clauses in $\mathcal{P}'$ (as given in appendix C) except for the clauses of form 4. To check soundness for these clauses suppose that $\mathcal{P}$ contains the clause $p(x) \leftarrow q(w)$ and that $ok(\Pi_x(\sigma,\ t))$, $=_w(t)$, and $q(t)$ are all $\mathcal{P}$-true. By the definition of $\mathcal{P}$-truth there exists a finite $\mathcal{P} \cup Q$-derivation $r$ of $q(D(t))$ such that all typed variables occuring in that derivation occur free in $t$. Let $\sigma(r)$ denote the derivation that results from replacing each free occurrence of a typed variable $var(c_y,\ \tau)$ in $r$ by $D(\sigma(y))$ and replacing each free occurence of atom of the form $p(var(c_y,\ \tau))$ in $r$ by a $\mathcal{P} \cup Q$-derivation of $p(D(\sigma(y)))$. The semantics of $ok$ guarantees that these latter derivation always exists. The semantics of $=_w$ guarantees that the the label in the root of $\sigma(r)$ is an instance of $q(w)$ where the value of $x$ is $D(\sigma(x))$. So by using the projection rule $p(x) \leftarrow q(w)$ we get a $\mathcal{P} \cup Q$-derivation of $p(D(\sigma(x))) = p(D(\Pi_x(\sigma,\ t)))$. We also have that all variables occuring in this derivation occur free some term of the form $\sigma(y)$ and hence occur free in in $\Pi_x(\sigma,\ \tau)$. So $p(\Pi_x(\sigma,\ t))$ is $\mathcal{P}$-true.

By the soundness of the rules in $\mathcal{P}'$ we have that If $p(t) \in [\![\mathcal{P}']\!]$ then there exists a finite $\mathcal{P} \cup Q$-derivation of $p(D(t))$ such that every typed variable in that derivation occurs free in $t$. In particular, we now have one half the desired equivalence between $\mathcal{P}$ and $\mathcal{P}'$ — if $t$ is a finite closed anntated term and $p(t) \in [\![\mathcal{P}']\!]$ then $p(D(t)) \in [\![\mathcal{P}]\!]$.

We now consider the issue of "completeness" for the program $\mathcal{P}'$. We do not have completeness in the traditional sense — it is not true that $p(t) \in [\![\mathcal{P}']\!]$ whenever $p(t)$ is $\mathcal{P}$-true. However, to prove that the emptyness of $[\![\mathcal{P}, p]\!]$ is equivalent to the existence of finite closed term $t$ in $[\![\mathcal{P}', p]\!]$ it suffices to prove a weaker form of completeness stating that there exists an "annotation operation" $A$ such that for any finite $t \in [\![\mathcal{P}, p]\!]$ we have that then $A(t)$ is a closed term with $D(A(t)) = t$ and $A(t) \in [\![\mathcal{P}', p]\!]$. Intuitively, the term $A(t)$ must be a "sufficiently annotated" version of $t$. The key idea is that contexts can be "stacked" so that a given term can be annotated with one context for each property that is proved by use of a projection rule. We can assume without loss of generality that all projection rules in $\mathcal{P}$ use the same variable $x$ on the left hand side. Given this assumption we introduce the notation $\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)$ where this notation abbreviates a term as defined by the following equations where $x$ is the variable used in the left hand sides of projection rules.

$$\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \langle\sigma_2,\ t_2\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)$$

$$= \Pi_x(\sigma_1[x := \Gamma(s,\ \langle\sigma_2,\ t_2\rangle\ \ldots,\ \langle\sigma_n,\ t_n\rangle)],\ t_1)$$

$$\Gamma(s) = s$$

In these equations $\sigma[x := t]$ abbreviates $subst(c_x,\ t,\ \sigma)$. The term $\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)$ represents a "tower of contexts" around the term $s$ — we can one context for each property of $s$ proved by a projection rule. Note that $D(\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)) = D(s)$. This annotation process can then be continued by adding annotations to the terms $t_i$, the values in the substituions $\sigma_i$, and proper subterms of $s$. A full proof that sufficient annotations can always be added is given later for the general afp case.

## 7.2 The gfp case

The gfp case is similar to the lfp case exacept that we must now handle infinite terms and infinite derivations. In the gfp case we consider infinite annotated terms generated by the grammar of the preceding section. In order for the predicates in $\mathcal{P}'$ to have the intended meaning over infinite terms we need to impose some well formedness conditions on the terms. We define the annotation height of a term of the form $var(c_x,\ \tau)$ or $f(t_1,\ \ldots,\ t_n)$ to be 0 and the annotation height of $\Pi_x(\sigma,\ t)$ to be one greater than the annotation height of $\sigma(x)$. Note that if $s$ has annotation height 0 then $\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)$ has annotation height $n$ — this term is a "stack" of $n$ contexts around the term $s$. However, since we allow infinite terms, there exist terms of infinite annotation height. A term $t$ will be called well formed if no subterm has infinite annotation height and $t$ does not contain infinitely long substitutions or any infinitely long lists in the types of typed variables. Note that the finite annotation height condition also ensures that the first equation in the definition of the deletion operator $D$ eventually terminates in one of the other equations and hence $D$ is well defined on all well-formed annotated terms. It is possible to construct an alternating-regular afp program that excepts exaclty the set of closed well-formed annotated terms. So to prove that emptyness of $[\![\mathcal{P}, p]\!]$ is decidable it now suffices to show that $[\![\mathcal{P}, p]\!]$ is nonempty if and only if there exists a closed well-formed annotated term $t$ such that $p(t) \in [\![\mathcal{P}']\!]$.

Soundness in the lfp case ensures that finite $\mathcal{P}'$-derivations can be translated into finite $\mathcal{P} \cup Q$-derivations. In the case of finite derivations this translations is done "bottom-up" — starting from the leaves of the $\mathcal{P}'$-derivation and moving up. In the gfp case, where infinite derivations must be handled, this translation is done "top-down" — starting at the root.

In the bottom-up lfp translation, substitutions are applied to $\mathcal{P} \cup Q$-derivations. Note that $\mathcal{P} \cup Q$-derivations have the pleasant property that they do not involve the "binding" construct $\Pi_x$. The top-down translation process must apply substitutions to $\mathcal{P}'$-derivations which may involve bindings. Consider a node in a $\mathcal{P}'$-derivation of the form $p(var(c_y,\ \tau))$. The typed variable at this node may or may not occur "free" in the overall derivation. We say that a position in a $\mathcal{P}'$-derivation is free in that derivation if it does not occur inside the second or third child of a use of a clause of form 4 or at or under any node labeled with a use of a predicate of the form $na_{p,x}$. An occurence of a variable $var(c_y,\ \tau)$ in a $\mathcal{P}'$-derivation is called free if it is a free occurence in the label at a free position in the derivation.

To precisely formulate an infinite top-down process of derivation transformation we first let $T$ be the infinite set of ground clauses of the form $p(D(t)) \leftarrow p(t)$ where $t$ is a closed annotated term. Although the clause set $T$ is not a program in the sense defined here, the notion of a derivation involving clauses from $T$ can be defined in the obvious way. We want to prove that any $\mathcal{P}'$ derivation of a closed atom $p(t)$ can be translated into a $\mathcal{P}$-derivation of $p(D(t))$. We can do this by an infinite rewriting process on $\mathcal{P} \cup T \cup \mathcal{P}'$-derivations. We will use the notation $[\Phi \leftarrow r_1, \ldots,\ r_n]$ to denote the derivation whose root is labeled with $\Phi$ and whose children are the derivations $r_1,\ \ldots,\ r_n$. If $B$ is a finite sequence of derivations then we write $[\Phi \leftarrow B]$ to denote the derivation whose root is labeled with $\Phi$ and whose children are the derivations in $B$. Given a $\mathcal{P}'$ derivation $[p(t) \leftarrow B]$ we first construct the $T \cup \mathcal{P}'$-derivation $[p(D(t)) \leftarrow [p(t) \leftarrow B]]$. The translation process is then defined by the following rewrite rules.

$$
\begin{array}{l}
[p(D(f(t_1,\ \ldots,\ t_n))) \leftarrow \\
\quad [p(f(t_1,\ \ldots,\ t_n)) \leftarrow \\
\quad\quad [q_1(t_1) \leftarrow B_1] \\
\quad\quad \vdots \\
\quad\quad [q_n(t_1) \leftarrow B_n]]] \\
\Rightarrow \\
[p(D(f(t_1,\ \ldots,\ t_n))) \leftarrow \\
\quad [q_1(D(t_1)) \leftarrow [q_1(t_1) \leftarrow B_1]] \\
\quad \vdots \\
\quad [q_n(D(t_1)) \leftarrow [q_n(t_n) \leftarrow B_n]]]
\end{array}
$$

$$
\begin{array}{l}
[p(D(\Pi_x(\sigma,\ t))) \leftarrow \\
\quad [p(\Pi_x(\sigma,\ t)) \leftarrow \\
\quad\quad [ok(\Pi_x(\sigma,\ t)) \leftarrow B_1]]] \\
\quad\quad [=_w\ (t) \leftarrow B_2] \\
\quad\quad [q(t_1) \leftarrow B_3] \\
\Rightarrow \\
[p(D(\sigma(x))) \leftarrow [p(D(\sigma(t))) \leftarrow \sigma(B_3)]]
\end{array}
$$

$$
\begin{array}{l}
[p(D(\Pi_x(\sigma,\ t))) \leftarrow [p(\Pi_x(\sigma,\ t)) \leftarrow [pr_{p,x}(\sigma) \leftarrow B]]] \\
\Rightarrow \\
[p(D(\sigma(x))) \leftarrow [p(\sigma(x)) \leftarrow B']]
\end{array}
$$

In the second rewrite rule the derivation sequence $\sigma(B_3)$ is define by replacing every free occurence of a typed variable of the form $var(c_x\ \tau)$ in the derivations in $B$ by $\sigma(x)$ and replacing the derivation of each free occurance an atom of the form $p(var(c_x,\ \tau))$ by a derivation of $p(\sigma(x))$. The semantics of the predicate $ok$ ensures that such a derivation exists. This substitution process does not introduce new free variables — every free variable in $\sigma(r)$ occurs free in a value in $\sigma$. If the initial atom $p(t)$ is such that $t$ is closed and well-formed then this rewriting process can be done in such a way that uses of clauses from $T$ and $\mathcal{P}'$ are driven ever deeper in the derivation and in the limit we have a $\mathcal{P}$-derivation of $p(D(t))$. So if there is a $\mathcal{P}'$-derivation of $p(t)$ where $t$ is closed and well-formed, there is also a $\mathcal{P}$-derivation of $p(D(t))$. This gives the "soundness half" of the desired result. We postpone the discussion of completeness to section 7.4.

### 7.3 The afp Case

The afp case is similar to the gfp case — rewrite rules are used to translate a $\mathcal{P}'$-derivation to a $\mathcal{P}$-derivation in the limit of infinite rewriting. In the afp case, however, we use the parity subject reduction theorem to prove that the limit of this rewriting process satisfies the parity condition. To use the parity subject reduction theorem we need a value scheme on well formed $\mathcal{P}'$-derivations that preserves priority. In order for the value map to preserve priority the "assumptions" in the typed variables must specify the maximum priority on the path to that assumption. For each predicate $p$ and priority $i$ occuring in $P$ we introduce a constant $c_{p,i}$ and define the syntax of annotated terms in the afp case as follows.

$$\tau ::= emptytype \mid type(c_{p,i},\ \tau)$$

$$t ::= \bot \mid var(c_x,\ \tau) \mid f(t_1,\ \ldots,\ t_n) \mid \Pi_x(\sigma,\ t)$$

$$\sigma ::= emptysubst \mid subst(c_x,\ t,\ \sigma)$$

This is identicle to the grammar of the lfp and gfp cases except for the use of constants $c_{p,i}$ rather than $c_p$ in the types of the typed variables. The notion of free and bound typed variables, closed and well-formed terms, and the deletion operator $D$ are defined as before. In the afp case the program $\mathcal{P}$ involves predicates of the form $p^i$ where $p$ is a predicate in $\mathcal{P}$ and $i$ is a priority occuring in $\mathcal{P}$. By a similar argument to that used in the gfp case, it suffices to construct a program $\mathcal{P}'$ such that for any predicate $p^i$ in $\mathcal{P}'$ we have that $[\![\mathcal{P}', p^i]\!]$ contains a closed well-formed annotated term if and only if $[\![\mathcal{P}, p]\!]$ is nonempty.

We now describe the program $\mathcal{P}'$ used in the afp case without giving the precise clauses. The complete afp program $\mathcal{P}'$ is given in appendix D. For each predicate $p$ and priority $i$ occuring in $\mathcal{P}$ we introduce the predicate $p^i$ with the same priority as $p$. For each regular clause

$$p(f(x_1,\ \ldots,\ x_n)) \leftarrow q_1(x_1),\ \ldots,\ q_n(x_n)$$

in $P$, and for each $i$ in $\mathcal{P}$, we include the following in $\mathcal{P}'$.

$$p^i(f(x_1,\ \ldots,\ x_n)) \leftarrow q_1^{\max[i,\ \alpha_{q_1}]}(x_1),\ \ldots,\ q_n^{\max[i,\ \alpha_{q_n}]}(x_n)$$

The program $\mathcal{P}'$ also contains a set of rules which can be used to derive $p^i(var(c_x,\ \tau))$ if and only if $\tau$ contains $c_{p,i}$. Note that the superscript does not influence the priority or "meaning" of the predicate $p^i$, it is simply used to keep track of the maximum priority occuring on a derivation path. Each "assumpiton" in a typed variable specifies the maximum priority that must exist on any path to that assumption. Now for each projection clause $p(x) \leftarrow q(w)$ in $\mathcal{P}$ occurring in $\mathcal{P}$ we add the following clauses to $\mathcal{P}'$.

$$p^i(\Pi_x(\sigma,\ t)) \leftarrow\ ok^i(\Pi_x(\sigma,\ t)),\ =_w(t),\ q^{\Omega(q)}(t) \quad (5)$$

The predicate $=_w$ is defined as before. The predicate $ok^i$ is analogous to the predicate $ok$. The program $\mathcal{P}'$ also includes "pass-the-buck" of the following form.

$$p^i(\Pi_x(\sigma,\ t)) \leftarrow pr^i_{p,0,x}(\sigma)$$

Atoms of the the form $pr^i_{p,j,x}(\sigma)$ are analogous to atoms of the form $pr_{p,x}(\sigma)$ used in the afp and gfp reductions except that the superscript $i$ keeps track of the maximum priority on the inference path and the subscript $j$ governs the priority of the predicate itself.

The notion of a free position in a $\mathcal{P}'$-derivation is defined as in the gfp case — a position in a $\mathcal{P}'$ derivation is free unless it occurs under the second or third child of a use of a clause of form 5 or at or under any node labeled with a use of a predicate of the form $na^i_{p,x}$ (analogous to the predicates $na_{p,x}$ in the lfp and gfp cases). The clauses in $\mathcal{P}'$ are all such that we have the following invariant.

$\mathcal{P}'$ **Path Invariant** If $p^j(u)$ occurs free in a $\mathcal{P}'$-derivation whose root is labeled with $p^i(t)$ then $j$ equals the maximum of $i$ and the maximum priority occuring on the path from the root to this occurence of $p^j(u)$.

We leave it to the reader to verify this invaraint by examining the rules in appendix D. Note, however, that in the projection rule of the form 5 all free positions occur under the antecedent $ok^i(\Pi_x(\sigma,\ t))$. To satisfy the above path invariant it is important that the superscript $i$ in the head transfers to this first antecedent.

A use of rule 5 is analogous to a let expression in the let terms of the previous section. Free occurances of the atoms of the form $p^j(var(c_y,\ \tau))$ in the derivation of $q^{\Omega(q)}(t)$ are analogous to variables in the body of a let expression. The value of these variables, i.e., the approriate derivation of $q(\sigma(x))$ is contained in the derivation of $ok^i(\Pi_x(\sigma,\ t))$. The inference rules ensure that if $t$ is well formed and contains a free occurence of $var(c_y,\ \tau)$ where $\tau$ contains $c_{p,j}$ then the derivation of $ok^i(\Pi_x(\sigma,\ t))$ contains a position $p$ with the label $p^{\max(i,j)}(\sigma(x))$ such that the maximum priority on the path from $ok^i(\Pi_x(\sigma,\ t))$ to this position is $j$. We leave it to the reader to verify that the implementation of $ok^i$ in appendix D has this property. Given this property of the implementation of $ok^i$ it is possible to define a priority-preserving value scheme on $\mathcal{P}'$-derivations analogous to the value scheme on let terms where uses of rule 5 are analogous to let expressions but where the value of a free occurance of $p^j(var(c_y,\ \tau))$ in the derivation of $q^{\Omega(q)}(t)$ is assigned a value equal to the appropriate occurance of $p^{\max(i,j)}(\sigma(y))$ in the derivation of $ok^i(\Pi_x(\sigma,\ t))$. In addition to the value assignments on uses of rule 5, we assign a value to nodes which are uses of the pass the buck rule. If $[p^i(\Pi_x(\sigma,\ t)) \leftarrow [pr^i_{p,0,x}(\sigma) \leftarrow B]]$ is a use of the pass the buck rule then the value of this node is the position in $B$ labeled with $p^i(\sigma(x))$.

To construct a translation process which is literally a $V$-reduction sequence we will also need to introduce "identity nodes". These are specially marked nodes in a derivation of the form $[\Phi \leftarrow [\Phi \leftarrow B]]$ or $[p(D(t)) \leftarrow [p(t) \leftarrow B]]$. An identity node is assigned as a value the first (and only) child position so that the rewrite step from $[\Phi \leftarrow [\Phi \leftarrow B]]$ to $[\Phi \leftarrow B]$ is a $V$-reduction. Let $I$ be the set of all identity clauses (this plays the role of $T$ in the gfp case). The values assigned in uses of the rule 5 and the pass-the-buck rule plus the value assigned to identity nodes define a continuous value scheme $V$ on all $I \cup \mathcal{P}'$-derivations.

We now procede in a manner directly analogous to the gfp case. Given a $\mathcal{P}'$-derivation of $p^i(t)$ satisfying the parity condition we first introduce an identity node at every node labeled with $p^i(t)$ so that $[p^i(t) \leftarrow B]$ is replaced by $[p^i(t) \leftarrow [p^i(t) \leftarrow B]]$. The result of this transformation also satisfies the parity condition. We then replace the label $p^i(t)$ of the root of the derivation by $p(D(t))$. We then use the following rewrite rules to transform the resulting $T \cup \mathcal{P}'$-derivation into a $\mathcal{P}$-derivation constructed as the limit of an infinite rewriting process.

$$
\begin{array}{l}
[p(D(f(t_1,\ \ldots,\ t_n))) \leftarrow \\
\quad [p^i(f(t_1,\ \ldots,\ t_n)) \leftarrow \\
\quad\quad [q_1^{\max(i,\Omega(q_1))}(t_1) \leftarrow [q_1^{\max(i,\Omega(q_1))}(t_1) \leftarrow B_1]] \\
\quad\quad \vdots \\
\quad\quad [q_1^{\max(i,\Omega(q_n))}(t_1) \leftarrow [q_n^{\max(i,\Omega(q_n))}(t_1) \leftarrow B_n]]]] \\
\Rightarrow \\
[p(D(f(t_1,\ \ldots,\ t_n))) \leftarrow \\
\quad [q_1(D(t_1)) \leftarrow [q_1^{\max(i,\ \Omega(q_1))}(t_1) \leftarrow B_1]] \\
\quad \vdots \\
\quad [q_n(D(t_1)) \leftarrow [q_n^{\max(i,\ \Omega(q_n))}(t_n) \leftarrow B_n]]]
\end{array}
$$

$$
\begin{array}{l}
[p(D(\Pi_x(\sigma,\ t))) \leftarrow \\
\quad [p^i(\Pi_x(\sigma,\ t)) \leftarrow \\
\quad\quad [ok^i(\Pi_x(\sigma,\ t)) \leftarrow B_1] \\
\quad\quad [=_w\ (t) \leftarrow B_2] \\
\quad\quad [q^{\Omega(q)}(t_1) \leftarrow [q^{\Omega(q)}(t_1) \leftarrow B_3]] \\
\Rightarrow \\
[p(D(\sigma(x))) \leftarrow []\ []\ [p(D(\sigma(t))) \leftarrow \sigma(B_3)]]
\end{array}
$$

$$
\begin{array}{l}
[p(D(\sigma(x))) \leftarrow [p^i(\Pi_x(\sigma,\ t)) \leftarrow [pr^i_{p,i,x}(\sigma(x)) \leftarrow B]]] \\
\Rightarrow \\
[p(D(\sigma(x))) \leftarrow [p^i(\sigma(x)) \leftarrow B']]
\end{array}
$$

In the second rule $\sigma(B_3)$ is defined as in the gfp case except that a free occurence of an atom of the form $p^j(var(c_y,\ \tau))$ is replaced by the appropriate derivation of $p^{\max(i,j)}(\sigma(y))$ in the derivation of $ok^i(\Pi_x(\sigma,\ t))$. The nodes of the form [] are priority zero childless nodes needed to match structure in the reduction as required by the technical definition of a $V$-reduction. It is not difficut to show that if $t$ is well formed, so that the above claims for derivations of $ok^i(\Pi_x(\sigma,\ t))$ hold, then each of these rewrites rules is a $V$-reduction. If $t$ is closed then these rewrite rules never encounter a free variable. Hence there is an infinite sequence of ever-deeper reductions whose infinite limit can be converted to a $\mathcal{P}$-derivation of $p(D(t))$ by deleting nodes of the form []. By the continuity of $V$-reductions and the parity subject reduction theorem we now have that the resulting $\mathcal{P}$-derivation also satisfies the parity condition. So we have proved that if there exists a parity-satisfying $\mathcal{P}'$-derivation of $p^i(t)$ where $t$ is closed and well-formed and then there exists a parity-satisfying $\mathcal{P}$-derivation of $p(D(t))$. We address completeness in the next section.

## 7.4 Completeness

We now prove that the afp program $\mathcal{P}'$ is complete in the sense that there exists an annotation operator $A$ such that for any term $t$ constructed from $\bot$ and the function symbols occuring in $\mathcal{P}$ we have that $A(t)$ is a closed well formed annotated term with $D(A(t)) = t$ and if $p(t) \in [\![\mathcal{P}]\!]$ then $A(t) \in [\![\mathcal{P}', t]\!]$.

The memorylessness property of afp logic programs implies that there exists a choice function of parity-satisfying $\mathcal{P}$-derivations. More specifically, there exists a choice function $c$ that for any atom $\Phi \in [\![\mathcal{P}]\!]$ we have that $c(\Phi)$ is a pair $\langle H \leftarrow B,\ \rho\rangle$ where $H \leftarrow B$ is a clause in $\mathcal{P}$ and $\rho$ a substitution defined on the variables of the clause such that $\rho(H) = \Phi$ and such that for any $\Phi \in [\![\mathcal{P}]\!]$ the $\mathcal{P}$-derivation specified by $c$ satisfies parity. Let $c$ be such a choice function.

Recall that $\Gamma(s,\ \langle\sigma_1,\ t_1\rangle,\ \ldots,\ \langle\sigma_n,\ t_n\rangle)$ denotes a tower of annotations above the term $s$. We define a $\mathcal{P}$-term to be a term constructed entirely from the function symbols occuring in $\mathcal{P}$. We define a $\mathcal{P}$-substitution to be a substitution mapping a finite set of variables to $\mathcal{P}$-terms. We define a simple annotated term to be an annotated term which does not contain any occurences of the functions of the form $\Pi_x$. Alternatively, a simple annotated term can be defined to be a term constructed from the functions in $\mathcal{P}$ plus typed variables of the form $var(c_x,\ \tau)$. We say there exists a $c$-$j$-path from $\Phi$ to $\Psi$ if $\Phi \in [\![\mathcal{P}]\!]$ and there exists a path in the $\mathcal{P}$-derivation of $\Phi$ specified by the choice function $c$ from the root to a position labeled with $\Psi$ where $j$ is the maximum priority on that path. For $p(t) \in [\![\mathcal{P}]\!]$ we say thta $p$ is a regular property of $t$ if $c(p(t))$ involves a regular rule and we say that $p$ is an projection property of $t$ if $c(p(t))$ involves an projection rule. To handle the recursive cases in the definition of $A$ we need to define $A$ to take a pair of a simple term and a $\mathcal{P}$-substitution. Now for any simple annotated term $t$ and $\mathcal{P}$-substitution $\rho$ defined on the variables in $t$ we define $A(t,\ \rho)$ as follows.

$$A(u,\ \rho) = \Gamma(A_0(u,\ \rho),\ \langle\sigma_1, s_1\rangle,\ \ldots,\ \langle\sigma_k,\ s_k\rangle)$$

$$A_0(f(s_1,\ \ldots,\ s_n),\ \rho) = f(A(s_1,\ \rho),\ \ldots,\ A(s_1,\ \rho))$$

$$A_0(var(c_y,\ \tau),\ \rho) = var(c_y,\ \tau)$$

In the first equation there is one pair $\langle\sigma_i,\ t_i\rangle$ for each projection property of $\rho(u)$. To define these pairs we first define $A(t)$ for a $\mathcal{P}$-term $t$ to be $A(t,\ \emptyset)$ where $\emptyset$ denotes the totally undefined substitution. Also note that by the definition of $\Gamma$ the substitution $\sigma_i$ need not provide a value for $x$ — the value for $x$ is always the first argument to $\Gamma$. Now if $c(p(u)) = \langle p(x) \leftarrow q(w),\ \gamma\rangle$ then there is a pair $\langle\sigma_i,\ s_i\rangle$ in the annotations in $A(u,\ \gamma)$ where $\sigma_i$ and $t_i$ are defined as follows. The substitution $\sigma_i$ is defined on variables $y$ in $w$ other than $x$ so that $\sigma_i(y)$ is $A(\gamma(y))$. The term $s_i$ is $A(w',\ \gamma)$ where $w'$ is the result of replacing each variable $y$ in $w$ (including $x$) by $var(c_y,\ \tau)$ where $\tau$ is the set of all constants $c_{q,j}$ such that $q$ is a regular property of $\gamma(y)$ and there exists a $c$-$j$-path from $p(\gamma(w))$ to $q(\gamma(y))$. For any simple term $t$ and $\mathcal{P}$-substitution $\rho$ we have that $A(t,\ \rho)$ is well-formed and that $D(A(t)) = t$. For any $\mathcal{P}$-term $t$ we have that $A(t)$ is closed.

We define an open $\mathcal{P}'$-derivation to be a tree $r$ in which each position $p$ is labeled with an atom $r(p)$ and where for each $p$ with children $C(p)$ in $r$ we have that either $r(p) \leftarrow r(C(p))$ is an instance of a clause in $\mathcal{P}'$ or $p$ is a leaf of $r$. The leaves of $r$ which are not instance of clauses in $\mathcal{P}'$ will be called premises of the open derivation. We say that a pair $\langle p^i(u),\ \rho\rangle$ is well-formed if $u$ is a simple term, $\rho$ is $\mathcal{P}$-substitution and $p$ is a predicate in $\mathcal{P}$ such that $p(\rho(u)) \in [\![\mathcal{P}]\!]$ and for every variable $var(c_y,\ \tau)$ in $u$ we have that if $q$ is a regular property of $\rho(y)$ and there exists a $c$-$j$-path from $p(\rho(u))$ to $q(\rho(y))$ then $\tau$ contains $c_{q,\max(i,\ j)}$. For any well formed pair $\langle p^i(t),\ \rho\rangle$ we define $A(p^i(t),\ \rho)$ to be $p^i(A_0(t,\ \rho))$ if $p$ is a regular property of $t$ and $p^i(A(t,\ \rho))$ if $p$ is an projection property of $t$. Note that if $p$ is a regular property of $u$ then the pass-the-buck rules always allow one to derive $p^i(A(u,\ \rho))$ from $p^i(A_0(u,\ \rho))$.

We now show that for any well formed pair $\langle p^i(u),\ \rho\rangle$ there exists a finite open $\mathcal{P}'$-derivation $r$ whose root is labeled with $A(p^i(u),\ \rho)$ and whose premises are all of the form $A(q^k(s),\ \gamma)$ where $\langle q^k(s),\ \gamma\rangle$ is well-formed and if the path from the root of $r$ to the premise $q^k(A(s,\ \gamma))$ has maximum priority $j$ then there exists a $c$-$j$-path from $p(\rho(u))$ to $q(\gamma(s))$. First consider a well-formed pair of the form $\langle p^i(u),\ \rho\rangle$ where $p$ is a regular property of $u$. In this case $A(p^i(u),\ \rho)$ equals $p^i(A_0(u,\ \rho))$. There are two subcases. First suppose $u = var(c_y,\ \tau)$. Since we allow $c$-$i$-paths of length 1 the definition of a well-formed pair implies that $\tau$ must contain $c_{p,i}$. In this case there is a finite derivation of $p^i(A_0(u,\ \rho)) = p^i(var(c_y,\ \tau))$ not involving any premises. Now suppose $u = f(s_1,\ \ldots,\ s_n)$. Since $p$ is a regular property of $\rho(f(s_1,\ \ldots,\ s_n))$ one can use a version of the same regular rule possibly followed by uses of pass-the-buck rules to derive $p^i(A_0(f(s_1,\ \ldots,\ s_n),\ \rho))$ from $A(q_1^{\max(i,\Omega(q_1))}(s_1),\ \rho),\ \ldots,\ A(q_n^{\max(i,\Omega(q_n))}(s_n),\ \rho)$. This results in a finite open $\mathcal{P}'$-derivation with premises of the form $A(q^k(s_i),\ \rho)$ where $\langle q^k(s_i),\ \rho\rangle$ is well formed and the required $c$-$j$-path exists. Finally, consider a well-formed pair of the form $\langle p^i(u),\ \rho\rangle$ where $p$ is an projection property of $u$, i.e., $c(p(u)) = \langle p(x) \leftarrow q(w),\ \gamma\rangle$. In this case we have $A(p^i(u),\ \rho) = p^i(A(u,\ \rho))$. The pass-the-buck rules can be used to reach a subterm of $A(u,\ \rho)$ of the form $\Pi_x(\sigma,\ A(w',\ \gamma))$ where $\sigma(x)$ is a partially annotated version of $u$ and for $y$ other than $x$ in $w$ we have that $\sigma(y)$ is $A(\gamma(y))$ and $w'$ is the result of replacing typed variables for variables in $w$ as specified above. The $\mathcal{P}'$ version of the projection rule $p(x) \leftarrow q(w)$ can be used to construct a finite open $\mathcal{P}'$-derivation of $p^i(\Pi_x(\sigma,\ A(w',\ \gamma)))$ involving a premise of the form $q^{\Omega(q)}(A(w',\ \gamma))$, premises of the form $q^k(A_0(\gamma(y)))$ where $q$ is a regular property of $\gamma(y)$ and $y \neq x$, and premises of the form $q^k(A_0(u,\ \rho))$ where $q$ is a regular property of $\rho(u)$ (handling the fact that $\sigma(x)$ is a paritally annotated version of $u$). It is possible to check that the premises correspond to well formed pairs and that the required $c$-$j$-paths exist.

Now by repeatedly replacing premises with finite open derivations we can construct a possibly infinite $\mathcal{P}'$-derivation (with no premises) of $A(p^i(u),\ \rho)$ for any well-formed pair $\langle p^i(u),\ \rho\rangle$. Any infinite path in this derivation corresponds to an infinite sequence of $c$-$j$-paths and therefore to some infinite path in a derivation specified by the choice function $c$. Hence the infinite $\mathcal{P}'$-derivations constructed in this way satsfy the parity condition. Note that if $u$ is a $\mathcal{P}$-term such that $p(u) \in [\![\mathcal{P}]\!]$ then $\langle p^i(u),\ \emptyset\rangle$ is well formed and hence there is a parity-satisfying $\mathcal{P}'$-derivation of $p^i(A(u))$ and we are done.

# 8 Open Problems

We finish the paper with three open problems. The first is to precisely characterize the sets definable by general afp logic programs. We have been able to show that gfp logic programs can express any set of finite terms contained in $\Sigma^1_1$ but the precise characterization for general afp programs remains open.

The second open problem concerns closure under complementation. For any gfp program progra $\mathcal{P}$ we have that $\neg[\![\mathcal{P}, p]\!]$ is the set of negative literals for which there exists parity-satsifying $\mathcal{P}$-coderivations. Unfortunately coderivations are infinitely branching and seem fundamentally different from derivations. The class of games defined by gfp programs is not self-dual. But the quenstion of closure under complementation for set definable by gfp logic programs remains open.

The third open problem concerns parity subject reduction. It seems likely that it possible to generalize the parity subject reduction theorem given here to infinite-term versions of the lambda calculus. The untyped lambda calculus does not seems likely to satisfy parity subject reduction because finite terms in which every node as priority 1 can transfinitely reduce to infinite terms. However, it should be possible to formulate an infinite-term version of the simply typed lambda calculus and such a calculus seems likely to satisfy parity subject reduction.

In conclusion, we have described a generalization of logic programming to arbitrary alternations of least and greatest fixed points and shown that nontrivial static analysis of such programs is possible.

# Appendix A: The Proof of Theorem 1

**Theorem**
$[\![\mathcal{P}]\!] = F^n_{\mathcal{P}}(\emptyset)$ *where $n$ is the largest priority of any predicate in $\mathcal{P}$.*

The proof is similar to the proof of Theorem 6.1 in [**?**] (or Theorem 3.2 in [**?**]). The proof is by induction on the priority $k$ in the operators $F^k_{\mathcal{P}}$. More specifically, for any program $\mathcal{P}$, integer $k$, and database $A$ we define a $k$-$\mathcal{P}$-$A$-derivation to be a tree $r$ where each node $w$ is labeled with a ground atom $r(w)$ and has children $C(w)$ and where for each node $w$ we either have that the priority of $r(w)$ is greater than $k$ and $r(w) \in A$ or the priority of $r(w)$ is no greater than $k$ and $r(w) \leftarrow r(C(w))$ is a ground instance of a clause in $\mathcal{P}$. A $k$-$\mathcal{P}$-$A$-derivation tree is said to satisfy parity if for every infinite path in that tree the largest priority occurring infinitely often on that path is even. We define $[\![\mathcal{P}]\!]_{k,A}$ to be the set of ground atoms $\Phi$ such that there exists a parity-satsfying $k$-$A$-$\mathcal{P}$-derivation of $\Phi$. We now show by induction on $k$ that for any data base $A$ we have $F^k_{\mathcal{P}}(A) = [\![\mathcal{P}]\!]_{k,A}$. The case for $k$ negative is trivial — for $k$ negative we have $\Phi \in F^k_{\mathcal{P}}(A)$ if and only if $\Phi \in A$ and there is a $k$-$\mathcal{P}$-$A$-derivation of $\Phi$ if and only if $\Phi \in A$. We now assume the theorem for $k$ and consider $k+1$ proving both $[\![\mathcal{P}]\!]_{k+1,A} \subset F_{\mathcal{P}}^{k+1}(A)$ and $F_{\mathcal{P}}^{k+1}(A) \subseteq [\![\mathcal{P}]\!]_{k+1,A}$ in each of the cases where $k+1$ even or odd.

First we prove $F_{\mathcal{P}}^{k+1}(a) \subseteq [\![\mathcal{P}]\!]_{k+1,A}$ in the case where $k+1$ is even. The fixed point definition of $F_{\mathcal{P}}^{k+1}(A)$ implies the following.

$$F_{\mathcal{P}}^{k+1}(A) = T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1}(A)))$$

By the induction hypothesis we have the following.

$$F_{\mathcal{P}}^{k+1}(A) = T_{\mathcal{P}}^{k+1}([\![\mathcal{P}]\!]_{k,A[k+1:=F_{\mathcal{P}}^{k+1}(A)]}$$

Now suppose that $\Phi \in F_{\mathcal{P}}^{k+1}(A) = T_{\mathcal{P}}^{k+1}([\![\mathcal{P}]\!]_{k,A[k+1:=F_{\mathcal{P}}^{k+1}(A)]}$. For any such $\Phi$ we have either that $\Phi$ has priority greater than $k+1$ and $\Phi \in A$ or there exists a nontrivial derivation of $\Phi$ every proper subderivation of which is in $[\![\mathcal{P}]\!]_{k,A[k+1:=F_{\mathcal{P}}^{k+1}(A)]}$. These subderivations have the property that every premise of priority $k+1$ is (recursively) in $F_{\mathcal{P}}^{k+1}(A)$. So it is possible to By repeatedly replacing premises of priority $k+1$ by nontrivial derivations. Repeatedly making such replacements results in a $k+1$-$A$-$\mathcal{P}$-derivation This derivation has the property that every infinite path only involves priorities of $k+1$ or less and is either eventually contained entirely in some $k$-$\mathcal{P}$-$A[k+1 := F_{\mathcal{P}}^{k+1}(A)]$-derivation or contains an infinite number of occurences of priority $k+1$. In either case the infinite path satisfies the parity condition. So we get that $\Phi \in [\![\mathcal{P}]\!]_{k+1,A}$.

Now we prove $[\![\mathcal{P}]\!]_{k+1,A} \subseteq F_{\mathcal{P}}^{k+1}(a)$ in the case where $k+1$ is even. If $k+1$ is even we have the following.

$$F_{\mathcal{P}}^{k+1}(A) = \nu B.\ T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := B]))$$

We now define $F_{\mathcal{P}}^{k+1,\alpha}(A)$ as follows.

$$F_{\mathcal{P}}^{k+1,\alpha}(A) = \cap_{\beta<\alpha} T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1,\beta}(A)]))$$

This definition yields the following.

$$F_{\mathcal{P}}^{k+1}(A) = \cap_{\alpha} F_{\mathcal{P}}^{k+1,\alpha}(A)$$

We prove by transfinite induction on $\alpha$ that for any $\Phi \in [\![\mathcal{P}]\!]_{k+1,A}$ we have $\Phi \in F_{\mathcal{P}}^{k+1,\alpha}(A)$. Assume the desired result for all $\beta < \alpha$ and consider an atom $\Phi \in [\![\mathcal{P}]\!]_{k+1,A}$. The induction hypothesis implies that for any $\beta < \alpha$ we have $\Phi \in T_{\mathcal{P}}^{k+1}([\![\mathcal{P}]\!]_{k,A[k+1:=F_{\mathcal{P}}^{k+1,\beta}(A)}$. By the induction hypothesis on $k$ we then have $\Phi \in T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1,\beta}(A))$. By the definition of $F_{\mathcal{P}}^{k+1,\alpha}(A)$ we then have $\Phi \in F_{\mathcal{P}}^{k+1,\alpha}(A)$.

Next we prove $F_{\mathcal{P}}^{k+1}(a) \subseteq [\![\mathcal{P}]\!]_{k+1,A}$ in the case where $k+1$ is odd. If $k+1$ is odd we have the following.

$$F_{\mathcal{P}}^{k+1}(A) = \mu B.\ T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := B]))$$

We now define $F_{\mathcal{P}}^{k+1,\alpha}(A)$ as follows.

$$F_{\mathcal{P}}^{k+1,\alpha}(A) = \cup_{\beta<\alpha} T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1,\beta}(A)]))$$

This gives the following.

$$F_{\mathcal{P}}^{k+1}(A) = \cup_{\alpha} F_{\mathcal{P}}^{k+1,\alpha}(A)$$

We now prove by transfinite induction on $\alpha$ that $F_{\mathcal{P}}^{k+1,\alpha}(a) \subseteq [\![\mathcal{P}]\!]_{k+1,A}$. Assume the result for all $\beta < \alpha$ and consider $\Phi \in F_{\mathcal{P}}^{k+1,\alpha}(A)$. By the definition of $F_{\mathcal{P}}^{k+1,\alpha}(A)$ we now have the following for some $\beta < \alpha$.

$$\Phi \in T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1,\beta}(A)]))$$

By the induction hypothesis on $k$ we then have the following.

$$\Phi \in T_{\mathcal{P}}^{k+1}([\![\mathcal{P}]\!]_{k,\ A[k+1:=F_{\mathcal{P}}^{k+1,\beta}(A)]})$$

Now by the induction hypothesis on $\beta$ there exists a parity-preserving $k+1$-$A$-$\mathcal{P}$-derivation of $\Phi$.

Finally we prove $[\![\mathcal{P}]\!]_{k+1,A} \subseteq F_{\mathcal{P}}^{k+1}(a)$ in the case where $k+1$ is odd. The fixed point definition of $F_{\mathcal{P}}^{k+1}(A)$ implies the following.

$$F_{\mathcal{P}}^{k+1}(A) = T_{\mathcal{P}}^{k+1}(F_{\mathcal{P}}^{k}(A[k+1 := F_{\mathcal{P}}^{k+1}(A)))$$

By the induction hypothesis on $k$ we have the following.

$$F_{\mathcal{P}}^{k+1}(A) = T_{\mathcal{P}}^{k+1}([\![\mathcal{P}]\!]_{k,A[k+1:=F_{\mathcal{P}}^{k+1}(A)]}$$

Now let $r$ be any parity-satisfying $k+1$-$A$-$\mathcal{P}$-derivation. The subnode relation on positions in $r$ of rank $k+1$ must be well founded — otherwise there would be an infinite path whose maximum infinite priority is $k+1$. We assign each node $p$ of priority $k+1$ in $r$ a rank equal to the least ordinal greater than the rank of all nodes of priority $k+1$ below $p$. The rank of $r$ is the least ordinal greater than the rank of all non-root positions of priority $k+1$ in $r$. We now prove by transfinite induction on $\alpha$ that if $\Phi$ has a $k+1$-$A$-$\mathcal{P}$-derivation of rank $\alpha$ then $\Phi \in F_{\mathcal{P}}^{k+1}(A)$. Assume the result for all $\beta < \alpha$ and suppose that $\Phi$ has a parity-preserving $k+1$-$A$-$\mathcal{P}$-derivation $r$ of rank $\alpha$. By the induction hypothesis on $\alpha$ we have that the label of every non-root node $q$ in $r$ of priority $k+1$ is contained in $F_{\mathcal{P}}^{k+1}(A)$. Therefore we have that $\Phi$ is contained in $T_{\mathcal{P}}^{k+1}(\llbracket\mathcal{P}\rrbracket_{k,A[k+1:=F_{\mathcal{P}}^{k+1}(A)]} = F_{\mathcal{P}}^{k+1}(A)$.

# Appendix B: Parity Subject Reduction Proofs

**Transitivity of $V$-redicibility** We now prove that for any value scheme $V$ the relation of $V$-reducibility is transitive. Let $f$ be a reduction map from $s$ to $r$ and $g$ a reduction map from $w$ to $s$. Note since $f(V_s(p)) \in V_r^*(f(p))$ we get that $f(V_s^*(p)) \subseteq V_r^*(f(p))$. We can now show that $f \circ g$ is a reduciton map from $w$ to $r$ by the following calculations.

$$\Omega_r(f(g(p))) = \Omega_s(g(p)) = \Omega_w(p)$$
$$f(g(V_w(p))) \in f(V_s^*(g(p))) \subseteq V_r^*(f(g(p)))$$
$$f(g(p.m)) \in f(V_s^*(g(p).m)) \subseteq V_r^*(f(g(p).m))$$
$$\subseteq V_r^*(V_r^*(f(g(p)).m)) = V_r^*(f(g(p)).m)$$

**Continuity of $V$-reducibility** We now prove that a limit of $V$-reductions is also a $V$-reduction. To prove this we will need a slightly stronger notion of limit that includes the reduciton maps as well as the term structure. We say that $r$ $V$-reduces to $s$ at depth $d$ written $r \stackrel{d}{\to} s$ if there exists a reduction map $f$ from $s$ to $r$ such that for all positions $p$ of depth less than $d$ we have that $r(p) = s(p)$ and $f(p) = p$. We define an ever-deeper reduction sequence to be an infinite reduction chain $r_1 \stackrel{d_1}{\to} r_2 \stackrel{d_2}{\to} r_3 \stackrel{d_3}{\to} \ldots$ such that for all $k > 0$ then there exists an $i$ such that $d_j > k$ for all $j > i$. It is easy to see that the trees $r_i$ in an ever-deeper reduction sequence must converge to a limit tree $r$. We now prove if $r$ is the limit of an every-deeper reduction chain $r_1$, $r_2$, $r_3$, $\ldots$ then $r_1$ $V$-reduces to the limit $r$. By the continuitive of the value scheme $V$ we have that limit $r$ is $V$-well-formed. The fact that reduction maps are closed under composition implies that there exists a reduciton map $f_i$ from each tree $r_i$ to $r_1$. The fact that the reductions are ever-deeper implies that the sequence of reduction maps $f_1$, $f_2$, $f_3$, $\ldots$ converges (in the same sense as for trees) to a map $f$. Now let $p$ be any position in the limit $r$ and let $p.m$ be any child of $p$ in $r$. Let $i$ be sufficiently large that $r(p) = r_i(p)$, $f(p) = f_i(p)$, $f(p.m) = f_i(p.m)$ and $V_r(p) = V_{r_i}(p)$. The convergence of $r_i$ and $f_i$ plus the continuity of $V$ ensure that such an $i$ exists. We then have the following.

$$\Omega_{r_1}(f(p)) = \Omega_{r_1}(f_i(p)) = \Omega_{r_i}(p) = \Omega_r(p)$$
$$f(V_r(p)) = f_i(V_{r_i}(p)) \in V_{r_1}^*(f_i(p)) = V_{r_1}^*(f(p))$$
$$f(p.m) = f_i(p.m) \in V_{r_1}^*(f_i(p).m) = V_{r_1}^*(f(p).m)$$

**Theorem 7 (Parity Subject Reduction)**
*If $V$ is a continuous value scheme on $L$-trees and $r$ is a $V$-well-formed tree satisfying the parity condition, if $r$ $V$-reduces to $s$ then $s$ satisfies the parity condition.*

First we define the notion of a value path in a $V$-well-formed tree. A value path is a sequence of positions $\alpha_1$, $\alpha_2$, $\alpha_3$, $\ldots$ such that $\alpha_{i+1}$ is either $V_r(\alpha_i)$ or $\alpha_i.m$ for some $m$. We first prove that for any infinite path in $s$ there exists an infinite value path in $r$ in which the maximum priority occuring infinitely often is the same. To see this let $\beta_1$, $\beta_2$, $\beta_3$ be an infinite path in $s$. Now consider the infinite sequence of positions in $r$ of the form $f(\beta_1)$, $f(\beta_2)$, $f(\beta_3)$, $\ldots$. By the definition of a reduction map, $f$ preserves priority and hence this infinite sequence has the same maximum infinitely occuring priority. Furthermore we have that $\beta_{i+1} = \beta_i.m$ for some $m$ so by the definition of a reduction map we have $f(\beta_{i+1}) = V_r^k(f(\beta_i).m)$ for some $k$. We can now define a new infinite sequence of $r$ positions by replacing each element $f(\beta_{i+1})$ by the finite sequence $f(\beta_i).m$, $V_r(f(\beta_i).$, $\ldots$, $V_r^k(f(\beta_i).m)$. Since the value map $V_r$ preserves priorities we have that each element in this finite sequence has the same priority as $f(\beta_{i+1}) = V_r^k(f(\beta_i).m)$. so the replacement does not change the maximum infinitely occuring priority. The replacement yields the desired value value path in $r$. It now suffices to show that for any infinite value path in $r$ the maximum infinitiely occuring priority is even.

Let $\alpha_1$, $\alpha_2$, $\alpha_3$, $\ldots$ be any value-path in $r$. We have that $\alpha_j$ is below or left of $\alpha_i$ for any $J > i$. Let $S$ be the set of all positions above or equal to some position of the form $\alpha_i$. A position $p$ in $S$ will be called $S$-leftmost if no position in $S$ is left of $p$. Note that the root position is $S$-leftmost. Furthermore, since the sequence $\alpha_i$ is infinite and moves strictly down or left, if $p$ is an $S$-leftmost element of $S$ then $S$ must contain some child of $p$ and the leftmost child of $p$ in $S$ must also be $S$-leftmost. So the $S$-leftmost positions in $S$ form an infinite descending chain.

We now prove that every $S$-leftmost position in $S$ is above some $S$-leftmost position of the form $\alpha_i$ and therefore that the infinite sequence $\alpha_i$ contains as a subsequence an infinte descending chain. Let $p$ be any $S$-leftmost position in $S$. We say that a position $p \in S$ is $S$-branching if $S$ contains at least two children of $p$. If there are no $S$-branching positions below or equal to $p$ then the positions in $S$ below $P$ form a descending chain all of which are $S$-leftmost and the result follows. If there exists some branching position below or equal to $p$ then there is a unique shallowest such position which must also be $S$-leftmost. So we can assume without loss of generality that $p$ is $S$-branching. Since the sequence of positions $\alpha_i$ moves monotonically down or left the set of positions of the form $\alpha_i$ which are below some child of $p$ other than the leftmost child must now be nonempty and finite. Let $i$ be the largest index such that $\alpha_i$ is below some child of $p$ other than the leftmost child. Since $\alpha_{i+1}$ is not below $\alpha_i$ we must have $\alpha_{i+1} = V_r(\alpha_i)$. We now must have that $p = V_r(\alpha_i) \vee \alpha_i$ and hence $p$ must be a binding position. By the locally-binding propery of value maps there can be no other binding node between $p$ and $V_r(\alpha_i) = \alpha_{i+1}$. Since $p$ is $S$-leftmost and $\alpha_{i+1} = V_r(\alpha_i)$ is below the leftmost child of $p$ we have that $\alpha_{i+1}$ must be $S$-leftmost. So the sequence $\alpha_1, \alpha_2, \alpha_3, \ldots$ must contain an infinite subsequence of $S$-leftmost positions.

It now suffices to prove that if $\alpha_i$ and $\alpha_j$ are both $S$-leftmost with $j > i$ then the maximum priority on the path from $\alpha_i$ to $\alpha_j$ in $r$ is the same as the maximum priority on the subsequence $\alpha_i, \alpha_{i+1}, \ldots, \alpha_j$. Note that if $\alpha_i$ is $S$-leftmost then all positions of the form $\alpha_j$ for $j > i$ are below $\alpha_i$. So it suffices to prove that $\alpha_i$ is $S$-leftmost then for all $j > i$ the path from $\alpha_i$ to $\alpha_j$ has the same maximum priority in $r$ as the subsequence $\alpha_i, \alpha_{i+1}, \ldots$ $\alpha_j$. The prove is by induction on $j$. The case for $j = i$ is trivial. Now suppose that the statement holds for $j$ and consider $j + 1$. If $\alpha_{j+1} = \alpha_j.m$ then the result follows immediately from the induction hypothesis that the path from $\alpha_i$ to $\alpha_j$ has the same maximu priority as the sequence $\alpha_{i+1}, \ldots\ \alpha_j$. Now suppose that $\alpha_{j+1} = V_r(\alpha_j)$. In this case, since $V_r$ preserves priority we have that $\alpha_{j+1}$ has the same priority as $\alpha_j$ so the sequence $\alpha_i, \alpha_{i+1}, \ldots$ $\alpha_j, \alpha_{j+1}$ has the same maximum priority as the sequence $\alpha_i, \alpha_{i+1}, \ldots\ \alpha_j$. By the induction hypothesis this has the same maximum priority as the path from $\alpha_i$ to $\alpha_j$. By the priority preservation property of value maps the path from $\alpha_i$ to $\alpha_j$ has the same maximum priority as the path from $\alpha_i$ to $V_r(\alpha_j) = \alpha_{j+1}$ and we are done.

# Appendix C: The program $\mathcal{P}'$ for the lfp and gfp cases

The program $\mathcal{P}'$ includes the regular rules in $\mathcal{P}$. For each projection clause $p(x) \leftarrow q(w)$ in $\mathcal{P}$ the program $\mathcal{P}'$ includes the following.

$$p(\Pi_x(\sigma,\ t)) \leftarrow\ ok(\Pi_x(\sigma,\ t)),\ =_w(t),\ q(t)$$

The program $\mathcal{P}'$ also includes all clauses of the following forms.

$$p(\Pi_x(\sigma,\ t)) \leftarrow pr_{p,x}(\sigma)$$

$$\begin{array}{l} p(var(c_x,\ \tau)) \leftarrow p(\tau) \\ p(type(c_p,\ \tau)) \leftarrow \\ p(type(c_q,\ \tau)) \leftarrow p(\tau) \end{array}$$

$$\begin{array}{l} =_y(\mathrm{var}(c_y,\ \tau)) \leftarrow \\ =_{f(w_1,\ \ldots,\ w_n)}(f(s_1,\ \ldots,\ s_n)) \leftarrow =_{w_1}(s_1),\ \ldots,\ =_{w_n}(s_n) \\ =_w(\Pi_x(\rho,\ t)) \leftarrow =_{x,w}(\rho) \\ =_{x,w}(subst(c_y,\ t,\ \rho)) \leftarrow =_{x,w}(\rho)\ \ \text{where } x \neq y \\ =_{x,w}(subst(c_x,\ t,\ \rho)) \leftarrow =_w(t) \end{array}$$

$$\begin{array}{l} ok(s) \leftarrow \{ok_{p,x}(s) :\ p, x \in \mathcal{P}\} \\ ok_{p,x}(\Pi_x(\sigma,\ t)) \leftarrow na_{p,x}(t) \\ ok_{p,x}(\Pi_x(\sigma,\ t)) \leftarrow pr_{p,x}(\sigma) \end{array}$$

$$\begin{array}{l} na_{p,x}(f(z_1,\ \ldots,\ z_n)) \leftarrow na_{p,x}(z_1),\ \ldots,\ na_{p,x}(z_n) \\ na_{p,x}(var(c_y,\ \tau)) \leftarrow\ \ \ x \neq y \end{array}$$

$na_{p,x}(var(c_x,\ \tau)) \leftarrow na_{p,x}(\tau)$
$na_{p,x}(type(c_q,\ \tau)) \leftarrow na_{p,x}(\tau) \quad p \neq q$
$na_{p,x}(emptytype) \leftarrow$
$na_{p,x}(\Pi_y(\sigma,\ t)) \leftarrow na_{p,x}(\sigma)$
$na_{p,x}(subst(c_y,\ t,\ \sigma)) \leftarrow na_{p,x}(t),\ na_{p,x}(\sigma)$

$pr_{p,x}(subst(c_x, t, \sigma)) \leftarrow p(t)$
$pr_{p,x}(subst(c_y, t, \sigma)) \leftarrow pr_{p,x}(\sigma)$ where $x \neq y$

# Appendix D: The program $\mathcal{P}'$ for the afp case

For each predicate $p$ in $\mathcal{P}$ and each priority $i$ occuring in $\mathcal{P}$ we have a predicate of the form $p^i$ whose priority is the same as $p$. For each pair of priorities $i$ and $j$ and each predcate $p$ and variable $x$ occuring in $\mathcal{P}$ we have a the predicate $pr^i_{p,j,x}$ of priority $j$. All other predicates have priority 0.

For each regular clause $p(f(x_1, \ \ldots, \ x_n) \leftarrow q_1(x_1),\ \ldots,\ q_n(x_n)$ in $\mathcal{P}$ we have that $\mathcal{P}'$ includes all clauses of the following form.

$$p^i(f(x_1,\ \ldots,\ x_n)) \leftarrow q_1^{\max(i,\Omega(q_1))}(x_1),\ \ldots,\ q_n^{\max(i,\Omega(q_n))}(x_n)$$

For each projection clause $p(x) \leftarrow q(w)$ in $\mathcal{P}$ the program $\mathcal{P}'$ includes all clauses of the following form.

$$p^i(\Pi_x(\sigma,\ t)) \leftarrow \ ok^i(\Pi_x(\sigma,\ t)),\ =_w(t),\ q^{\Omega(q)}(t)$$

The program $\mathcal{P}'$ also includes all clauses of the following forms.

$$p^i(\Pi_x(\sigma,\ t)) \leftarrow pr^i_{p,0,x}(\sigma)$$

$p^i(var(c_x,\ \tau)) \leftarrow p^i(\tau)$
$p^i(type(c_{p,i},\ \tau)) \leftarrow$
$p^i(type(c_{q,j},\ \tau)) \leftarrow p^i(\tau)$

The clauses defining $=_w$ as in appendix C.

$ok^i(s) \leftarrow \{ok^{i,j}_{p,x}(s) :\ i, j, p, x\ \in \mathcal{P}\}$
$ok^{i,j}_{p,x}(\Pi_x(\sigma,\ t)) \leftarrow na^j_{p,x}(t)$
$ok^{i,j}_{p,x}(\Pi_x(\sigma,\ t)) \leftarrow pr^{\max[i,j]}_{p,j,x}(\sigma)$

$na^i_{p,x}(f(z_1,\ \ldots,\ z_n)) \leftarrow na^i_{p,x}(z_1),\ \ldots,\ na^i_{p,x}(z_n)$
$na^i_{p,x}(var(c_y,\ \tau)) \leftarrow \quad x \neq y$
$na^i_{p,x}(var(c_x,\ \tau)) \leftarrow na^i_{p,x}(\tau)$
$na^i_{p,x}(type(c_{q,j},\ \tau)) \leftarrow na^i_p(\tau) \quad \langle p,\ i\rangle \neq \langle q,\ j\rangle$
$na^i_p(emptytype) \leftarrow$
$na^i_{p,x}(\Pi_y(\sigma,\ t)) \leftarrow na^i_{p,x}(\sigma)$
$na^i_{p,x}(subst(c_y,\ t,\ \sigma)) \leftarrow na^i_{p,x}(t),\ na^i_{p,x}(\sigma)$

$pr^i_{p,j,x}(subst(c_x, t, \sigma)) \leftarrow p^i(t)$
$pr^i_{p,j,x}(subst(c_y, t, \sigma)) \leftarrow pr^i_{p,j,x}(\sigma)$ where $x \neq y$