A Programming Calculus Based on Partial Inductive Definitions

with an introduction to the theory of partial inductive definitions

by

Lars-Henrik Eriksson and Lars Hallnäs

SICS R88013
Research Report
ISSN 0283-3638

CONTENTS

ACKNOWLEDGEMENTS

The authors wish to thank their colleagues at the Swedish Institute of Computer Science for their suggestions and for providing a stimulating research environment. In particular we want to thank Annika Wærn who read the manuscript in detail and suggested several improvements. Peter Dybjer read an earlier version of the report and gave several suggestions for clarification and improvement.

1. INTRODUCTION

Methods and principles for the specification, verification and synthesis of programs is an important field of computer science. Several different approaches have been developed for use with imperative, functional or logic programs. [16] contains a collection of several recent papers in the field, and Loeckx and Sieber [14] give a thorough presentation of traditional program verification. In the area of logic programming, there have been two main approaches, the transformational and the deductive approach. In the transformational approach (e.g. [12]), correctness preserving transformation rules are used to transform a specification into a program, or to show that the specification and the program has identical intentions. In the deductive approach, the correctness of the program is shown by derivations from the specification within a formal system. Examples of work using the deductive approach are the Logic Programming Calculus by Hansson, Tärnlund et.al. [3] [4] [8] and the work by Hogger [10].

The approach presented in the present article is based on one of the authors (L-H Eriksson) experiences with the Logic Programming Calculus (LPC). The LPC covers both verification, synthesis and transformation of logic programs, and has been a promising methodology. However, the LPC has never been given a firm theoretical base, and its criterion for program termination is in fact incorrect in general, although it works in practise. During attempts to give the LPC such a theoretical base major technical problems were encountered. It appeared that the reason for this was that programs and specifications were the same kind of entities and that there was a possibility of undesirable interaction between them in the derivations used to establish program correctness.

It was then suggested by the second author (Lars Hallnäs) that this problem could be solved by changing the relationship between specifications and programs and that the calculi of partial inductive definitions could be used as a base for the formal reasoning. With this approach it turned out that the problems encountered with the LPC could be solved easily.

Our approach relies heavily on partial inductive definitions. These are a generalization of the inductive definitions commonly used to define sets in terms of how elements of a set are constructed from other elements. The partial inductive definitions generate a special kind of consequence relation, similar in intention to logical consequence, but in general quite different in meaning. As in logic, we have deductive systems, similar to the sequent calculus, in which expressions can be inferred. Our presentation presumes a knowledge of mathematical logic.

As the theory of partial inductive definitions is not widely known, and published articles about them are quite technical in nature, we devote the first part of this article to an introduction to partial inductive definitions. The interested reader is referred to [1] and [6] for details.

The second part of the article is concerned with our method for program specification, verification and synthesis. A program, in this context, is basically a partial inductive definition. As sets of Horn clauses in logic (even extended with negation) can be seen as a special case of partial inductive definitions, our method can be used with logic programs. The abstract model of execution is, as in logic programming, that of "execution as deduction" – here a deduction in the calculi of partial inductive definitions.

Specifications need not be written in any fixed specification language. Any specification language for which the semantics is expressed using partial inductive definitions can be used, e.g. first-order predicate logic.

We give criteria for the partial correctness and termination and show that they are valid. These criteria are based on the derivability of certain expressions in the calculi of partial inductive definitions. Finally we outline a methodology to perform these derivations, which can be used both for verifying that a program is correct and to synthesize a correct program from a given specification.

This work is still in an early stage, and the present article is intended to lay down the foundations of our programming calculus. A complete methodology for program verification and synthesis using this approach remains to be developed.

2. INDUCTIVE DEFINITIONS

A definition of a set should permit us both to test for membership of the set, and to show that a property is shared by all members of the set. The purpose of an inductive definition [1] is to provide such definitions. An inductive definition specifies the members of a set in terms of other members of the same set. A typical example is:

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

Here x.l is **defined** to be a list if l is a list. We can easily see that a.$\varnothing$ is a list, but that a.b is not a list. a.b is defined to be a list if b is a list. Thus if b was a list, a.b would be too. However, there is nothing that defines b to be a list under any circumstances, so a.b cannot be a list.

We can formalize this slightly by writing:

$$\Rightarrow \varnothing$$
$$l \Rightarrow x.l, \quad \text{for all } x, l.$$

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

In this example we have used the pair "x.l", which is constructed from x and l. We do not want to concern ourselves with such construction processes, but rather regard the pair "x.l" as a unit. To emphasize this, we will call the entities used in the definitions "atoms". This means that there must be a separate expression defining every possible list, making the definition infinite in size. The line

$$l \Rightarrow x.l, \quad \text{for all } x, l.$$

should be regarded only as a finite description of how to generate the infinite number of expressions. To understand this description and to generate the expressions, we must have an understanding of how pairs are constructed, but once the expressions have been generated, we do not need this understanding to decide whether or not something is defined to be a list. We will be informal about how these constructions are actually done.

There are in general also atoms that do not occur in the definition. We call the totality of atoms a universe. We can now state the formal definition of an inductive definition.

DEFINITION 2.1 Clauses

Given a universe U of atoms, if $E \subseteq U$ and $e \in U$, then $E \Rightarrow e$ is a **clause** over U. e is called the **head** of the clause. The set of clauses over U will be denoted by clause(U). ■

DEFINITION 2.2 Inductive definitions

A set of clauses is an **inductive definition.** ■

Definitions 2.1 and 2.2 will be generalized in the next section.

In this section the following conventions will be used: We will assume the existence of a suitable universe and refer to it as U. The lower case letters a,b,c,d,e will be used to represent atoms, the upper case letters C and D will represent clauses, and the upper case letters E,F,G,H will represent sets of atoms. The letter P will commonly be used to represent inductive definitions. Other letters will represent unspecified entities, or the entities they represent will be given when they are used.

EXAMPLE 2.3

$\{\Rightarrow \varnothing\} \cup \{l \Rightarrow x.l \mid \forall x\ \forall l\}$ is an inductive definition of lists. ■

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

$$\frac{\{\vdash e' \mid e' \in E\}}{\vdash e} \vdash P \qquad \text{under the condition that } (E \Rightarrow e) \in P$$

The notation $\vdash e$ should be read "e holds". Should there be a possibility of confusion over which inductive definition is intended, we will suffix the actual definition used as in $\vdash_P e$, meaning "e holds according to P". The notation $\vdash_{P,P'}$ will mean the same as $\vdash_{P \cup P'}$. The notation $\vdash F$, where F is a set, will be short for "$\vdash e$, for all $e \in F$". For brevity, we will often say that "e is derivable", when we really mean that "$\vdash e$" is derivable.

We can see that by repeatedly using this inference rule, we can find a proof that a certain atom holds.

The "P" in the name of the inference rule ("$\vdash P$") does not refer to any particular inductive definition P. The reason "P" is used in the name of the inference rule is that "P" is commonly used as the name of inductive definitions. Nevertheless, the inference rule will still be called "$\vdash P$" regardless of the name of the inductive definition.

EXAMPLE 2.4

Using the inductive definition of example 2.3, we have:

$$\frac{}{\vdash \varnothing}\vdash P$$ since $\Rightarrow \varnothing$ is in the definition. No premises are needed here.

$$\frac{\vdash \varnothing}{\vdash a.\varnothing}\vdash P$$ since $\varnothing \Rightarrow a.\varnothing$ is in the definition. ■

The concepts of derivation, derivability, depth of a derivation etc. are defined in the same manner as in logic. A straightforward generalization is required when the left-hand side of a clause is infinite, since an inference rule with this clause would have to have infinitely many premises.

We can now formally characterize an inductively defined set, as one where all elements hold according to a certain inductive definition.

DEFINITION 2.5 Inductively defined sets

The set $Def(P) = \{e \mid \vdash_P e\}$ is the set **inductively defined** by P. ■

Letting P be the definition of lists, Def(P) would be the set of all lists.

In general, there could be several different proofs that some atom holds. It would sometimes be advantageous to have unique proofs for every atom. If every atom is defined by at most one clause, then an occurrence of the inference rule deriving that atom could have at most one set of premises, and the proof would be unique. We will formalize this "determinacy" property.

DEFINITION 2.6 Determinacy

If $(E \Rightarrow e) \in P$ and $(E' \Rightarrow e) \in P$ implies $E=E'$, for all e, then P is **deterministic.** ■

One advantage of having an inductive definition of a set is that an induction principle is immediately available for the set. Suppose that some property of atoms is preserved by the inference rule, i.e. if all premises have the property, then so does the conclusion. Since every element of an inductively defined set can be derived using applications of the inference rule, all of which preserves the property, every element must have the property. This is the principle of P-induction.

THEOREM 2.7 The principle of P-induction

Given a property R. Suppose that for all $(E \Rightarrow e) \in P$, R(e) holds if for all $e' \in E$, R(e') holds, then R(a) holds by **(P-)induction** for all $a \in Def(P)$.

PROOF.

Immediate, using induction over the derivation of $\vdash a$. ■

EXAMPLE 2.8

We show using induction that every non-empty list has a last element. Let R(a) be the property that a has a last element or is empty. We must show that this property is carried over by every clause in the definition.

The first clause is $\Rightarrow \varnothing$. R($\varnothing$) holds trivially.

The other clauses are of the form $l \Rightarrow x.l$. Suppose R(l), then either l is $\varnothing$ or it has a last element. In the first case x is a last element of x.l, in the other case x.l trivially has a last element, thus R(x.l).

Q.E.D. ■

In other presentations [1], it is usual to give an alternative definition of an inductively defined set. Since we deal with a proof-theoretic framework in this article, we believe that our definition is the one most easily understood. We will now state the alternate definition and show that they are equivalent.

DEFINITION 2.9 P-closedness

A set S is **P-closed** iff, for all $(E \Rightarrow e) \in P$, $E \subseteq S$ implies $e \in S$. ■

DEFINITION 2.10 $\models$

Let S be the smallest P-closed set. $\models$ **e** is defined to hold iff $e \in S$. ■

$\models F$ will be a shorthand for "$\models e$, for all $e \in F$".

THEOREM 2.11 Alternative formulation of inductively defined sets

Let S be the smallest P-closed set. Then **S = Def(P)**.

PROOF

1) From the inference rule, we can see that Def(P) is a P-closed set. Since S is the smallest P-closed set, $S \subseteq Def(P)$.

2) If all premises of an inference rule belongs to S, then, by definition, so does the conclusion. By P-induction, so does every $e \in Def(P)$, $Def(P) \subseteq S$ follows. ■

From this we can see that $\vdash$ and $\models$ are equivalent. Following standard practise, from now on we will use $\models$ when we are simply interested in membership in Def(P). $\vdash$ will be used when we are specifically interested in derivability using the inference rule.

We have given a clause "$e_1,\ldots,e_n \Rightarrow e$" the reading "if e_1 and ... and e_n hold, then e holds". By substituting "true" for "holds", we get a logical reading of the clause, namely "e_1 and ... and e_n implies e". With this reading a P-closed set is a model of the clauses in P and Def(P) is the smallest model. This way of reading a clause is **not** possible in general when we generalize the concept of a clause in the next section.

Note that with the logical reading, our clauses are precisely variable-free instances of those clauses that are called "Horn clauses" in logic. The generalized clauses need not have this form, so we will use the term "Horn clause" to refer to the kind of clause we have seen so far and "Horn clause definition" for an inductive definition consisting exclusively of Horn clauses.

3. PARTIAL INDUCTIVE DEFINITIONS

We will now generalize the concept of an inductive definition to that of a "partial inductive definition" [6]. A clause $E \Rightarrow e$ expresses the fact that e holds under the assumption that all $e' \in E$ hold. So far we have only been permitted to **state** such facts. However, one could imagine the possibility of having a "hypothetical" reasoning similar to hypothetical reasoning in logic, where we assume that certain atoms hold and **show** such a clause to be a fact.

To do this we permit assumptions to the left of the derivability symbol ($\vdash$), modify the $\vdash$P-rule to pass assumptions to the left, and add axioms of the form $e \vdash e$ to our calculi.

EXAMPLE 3.1
a.b.x holds (is a list) according to the inductive definition of lists, under the assumption that x holds (is a list).

$$\dfrac{\dfrac{x \vdash x}{x \vdash b.x}\vdash P}{x \vdash a.b.x}\vdash P \qquad \text{(an axiom)}$$

■

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

$$\dfrac{E \vdash e}{\vdash E \Rightarrow e}\vdash\Rightarrow$$

We can now complete the derivation:

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

For full generality, this conclusion should itself be permitted to be a premise to the $\vdash$P-rule. Since a condition for the applicability of the $\vdash$P rule is that there exist a clause $E \Rightarrow e$ in the definition, such that there is a premise $\vdash e'$ for each $e' \in E$, we then must also permit clauses to occur in the left hand side of other clauses. The depth of nesting in clause is called its level. We now define general clauses (also called Generalized Horn clauses in [7], or clauses of higher level).

DEFINITION 3.2 (General) clauses
Given a universe U of atoms, if E is a set, the elements of which are atoms or (general) clauses over U, and $e \in U$, then $E \Rightarrow e$ is a **(general) clause** over U. e is called the **head** of the clause. ■

Note that definition 3.2 is itself a Horn clause definition, permitting us to use induction over sets of general clauses.

DEFINITION 3.3 Levels

The **level** of an atom is 0. The **level** of a clause $E \Rightarrow e$ is

$$\begin{cases} 0 & \text{if } E \text{ is empty.} \\ \sup\{\text{level}(e') \mid e' \in E\}+1 & \text{otherwise.} \end{cases}$$

■

DEFINITION 3.4 Partial inductive definitions

A set of general clauses is a **partial inductive definition**. The **level** of a partial inductive definition is the supremum of the level of its clauses. ■

The reason for the name "partial inductive definition" will be explained later in this section.

Clearly, any Horn clause is also general clause and any Horn clause definition is also a partial inductive definition, while the converse in not true. In the sequel, we will use the term "clause" by itself to refer to general clauses.

Readers familiar with the theory of inductive definitions [1] or the fixpoint semantics of logic programs [13] might note that while the natural operator associated with a Horn clause definition is always monotone, this need not be the case for a partial inductive definition.

As we have generalized the notion of clauses, the conventions from section 2 need to be restated. In this section and section 4, the lower case letters a,b,c,d,e will be used to represent atoms, the upper case letters C and D will represent clauses or atoms (i.e. what can occur to the left of $\Rightarrow$ in a clause), and the upper case letters E,F,G,H will represent sets of clauses or atoms. The letter P will commonly be used to represent inductive definitions. U will still represent some suitable universe.

So far we have generalized the calculi with the $\vdash\Rightarrow$ inference rule and the axiom schema. We can also consider including clauses among the assumptions. If a set of atoms $e_1,\ldots,e_n$ are derivable and C is derivable from e, then C should be derivable from $e_1,\ldots,e_n \Rightarrow e$. This is analogous to the inference rule in sequent calculus for an implication among the assumptions.

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

In a sense, an application of the $\vdash$P rule corresponds to going from a definition to the atom being defined, i.e. if e is defined by the clause $E \Rightarrow e$, if we know E we can derive e. We complete the

extension of the calculi by adding an inference rule that permits us to go from an atom to its definition, i.e. to show that C is derivable under the assumption e, where e is defined by the clause $E \Rightarrow e$, we could show that C is derivable under the assumptions E. In case several different clauses could define e (if the definition was nondeterministic), we could not know which one actually did define it, consequently we would have to have one premise for each clause defining e. This is roughly analogous to or-elimination in natural deduction systems of logic.

$$\frac{\{E \vdash C \mid (E \Rightarrow e) \in P\}}{e \vdash C} P\vdash$$

Note especially that under the assumption of an atom e that could not possibly hold, i.e. for which there is no clause $E \Rightarrow e$ in the inductive definition defining it, then anything holds. This is a reasonable behavior since assuming something that could never hold is, in a sense, a contradiction. The behavior of our calculi in this case corresponds to that in logic, anything is derivable from a contradictory assumption.

We can now make a complete restatement of the calculi of inductive definitions, extended to handle partial inductive definitions. The inference rules and axiom schema are the same as above, with the extension that they all have to pass along assumptions that are not used in a particular inference.

$$\frac{\{F \vdash C' \mid C' \in E\}}{F \vdash e} \vdash P \quad \text{condition: } (E \Rightarrow e) \in P$$

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

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

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

$F,e \vdash e$ (axiom schema)

We regard the clauses to the left of the $\vdash$ symbol to form a set instead of a sequence, as is customary in sequent calculi. In this way we can freely reorder and duplicate assumptions and do without structural inference rules. The notation $\vdash F$, where F is a set, will be short for "$\vdash C$, for all $C \in F$".

A problem with performing derivations using these inference rules is that they become, in general, infinite. This is not a problem in principle, since they can be represented in a finite way (as long as the partial inductive definition determining the calculus can itself be represented in a finite way). We will not describe any general way of doing this, but in section 4 we will describe a simple finite representation for use in examples.

For each particular partial inductive definition, the relation $\vdash_P$ is monotone with respect to the assumptions, i.e. $F \vdash C$ implies $F,G \vdash C$. On the other hand, the partial inductive definitions themselves could be non-monotone, i.e we could have $\vdash_P e$ but $\nvdash_{P,P'} e$.

EXAMPLE 3.5

Let P be $(a \Rightarrow b) \Rightarrow c$. We have $\vdash c$, since

$$\dfrac{\dfrac{\dfrac{\dfrac{}{}\,P\vdash \quad \text{Since a is "absurd".}}{a \vdash_P b}\vdash\Rightarrow}{\vdash_P a \Rightarrow b}\vdash P}{\vdash_P c}$$

■

Now, let P' be $\Rightarrow a$. In $P \cup P'$, a would no longer be absurd, and the topmost inference would need the premise $\vdash b$. However, this cannot hold since b is not defined. As this derivation is the only possible, we get $\nvdash_{P,P'} c$.

Questions about extensions of partial inductive definitions such as this, and their properties, will be important to our programming calculus. We will define the terminology and properties involved.

DEFINITION 3.6 Extensions and restrictions

A partial inductive definition P' is called an **extension** to the partial inductive definition P, iff $P \subseteq P'$. Conversely P is a **restriction** of P'. ■

DEFINITION 3.7 Monotonicity of extensions

An extension P' of a partial inductive definition P is **monotone** with respect to a set of atoms $E \subseteq U$ iff, $\vdash_P e$ implies that $\vdash_{P'} e$, for all $e \in E$. ■

DEFINITION 3.8 Conservativity of extensions

An extension P' of a partial inductive definition P is **conservative** with respect to a set of atoms $E \subseteq U$ iff, $\vdash_{P'} e$ implies that $\vdash_P e$, for all $e \in E$. ■

From example 3.5, we can see that monotonicity of partial inductive definitions does not hold in general. Conservativity is, of course, even less likely to hold.

When dealing with Horn clause definitions, we drew parallels to logic and showed that the symbol $\Rightarrow$ could be interpreted as equivalent to implication. As we have seen, that parallel has its limits and $\Rightarrow$ can in general **not** be read as implication when we deal with partial inductive definitions. Although a

side look at such a reading, as we have done, can simplify understanding the intuition behind the inference rules, reliance on this reading will lead to serious mistakes. We will see further on, however, that for certain classes of definitions $\Rightarrow$ can be read as implication.

Note in particular, that the cut rule of the sequent calculus is not included in our calculi. This rule is redundant in (normal) logic and could be omitted from a logic sequent calculus. In the case of partial inductive definitions, such a rule would not be eliminable, so its omission is an essential one. Again, for certain classes of definitions the cut rule would hold. Another way of putting it is that the relation $\vdash$ is not in general transitive, i.e. $F \vdash G$ and $G \vdash C$ does not, in general, imply $F \vdash C$.

EXAMPLE 3.9

Let P be $(a \Rightarrow b) \Rightarrow a$. We have

$$\dfrac{\dfrac{\dfrac{\dfrac{a \vdash a \quad b \vdash b}{a, a \Rightarrow b \vdash b}\Rightarrow\vdash}{a \vdash b}\mathrm{P}\vdash}{\vdash a \Rightarrow b}\vdash\Rightarrow}{\vdash a}\vdash\mathrm{P}$$

■

So $\vdash a$ and $\vdash a \Rightarrow b$, but $\nvdash b$. We can see this as the only inference that could be used to derive $\vdash b$ would be $\vdash$P, but there is no clause defining b, so $\vdash$P would not be applicable.

Were there a cut rule, we could derive $\vdash b$ in this way:

$$\dfrac{\begin{array}{c}\vdots\\ \vdash a \Rightarrow b\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash a\end{array} \quad b \vdash b}{a \Rightarrow b \vdash b}\Rightarrow\vdash}{\vdash b}\mathrm{cut}$$

(the dots are parts of the derivation above)

So the omission of the cut rule is an essential one, which shows that $\Rightarrow$ cannot be read as implication.

This example is essentially the "liar" paradox. b can be read as absurdity, since it is not defined. The definition then states that a holds if and only if assuming that it holds leads to absurdity. From the definition we can see both that a holds ($\vdash a$) and that it would lead to absurdity ($\vdash a \Rightarrow b$), but from these contradictory statements we cannot draw the conclusion that anything holds, as we would if we tried to state the same fact in logic ($\neg a \leftrightarrow a$). Thus we can say that the calculi of partial inductive

definitions gives us a "local" notion of consequence, in the sense that further conclusions cannot be drawn from derived facts in an arbitrary way.

From these examples we see that the calculi of partial inductive definitions, while superficially similar to logic, are in fact much more general.

These properties makes the calculi of partial inductive definitions an interesting candidate for a formal base of knowledge-based systems as local inconsistencies cannot in general propagate beyond the particular spot where they occur. The non-monotonicity is also valuable in this context as it permits default reasoning.

Returning to our original aim of defining sets using inductive definitions, we define sets using partial inductive definitions in the same manner as for Horn clause definitions.

DEFINITION 3.10 Sets defined by partial inductive definitions
The set of **atoms** $\mathrm{Def}(P) = \{a \mid \vdash_P a\}$, is the set **inductively defined** by P. ∎

Although all sets can be defined by a Horn clause definition, it might be that such a definition is not a very natural one. A set can be trivially defined by simply including level 0 clauses defining every element in the set. Since the structure of a set can be arbitrarily complex, it might not be easy – or possible – to give an informal account of what elements would be defined. A partial inductive definition can then give a simpler definition.

Syntactic concepts, such as the set of lists, or of propositions, are natural to express by Horn clause definitions, as they are in general best defined by how the elements of the set are constructed. Semantic notions, on the other hand, are often more naturally expressed by definitions of higher level.

EXAMPLE 3.11
Consider the fragment of propositional logic containing only implication. The set of all propositions is naturally given by the following Horn clause definition:

$\Rightarrow A$	for each propositional variable A.
$A,B \Rightarrow A \rightarrow B$	for every A and B ($A \rightarrow B$ is a proposition if A and B are).

The semantics could be expressed by a level 1 (Horn clause) definition in the following way:

$\Rightarrow A$	for each A which is an instance of an (Hilbert) axiom schema for the propositional calculus).
$A, A \rightarrow B \Rightarrow B$	for every A and B (modus ponens).

On the other hand, the following level 2 definition would be more natural:

$A \Rightarrow A$	for each propositional variable A.
$(A \Rightarrow B) \Rightarrow A \rightarrow B$	for every A and B (A implies B if B follows from A).

■

The first clauses ($A \Rightarrow A$) in the last definition of example 3.11 are intended to say that we know nothing about the truth and falsity of propositional variables. In this way we cannot say that A holds, since it is defined in terms of itself. On the other hand, we cannot say that it is absurd either, since there is a clause defining it. This construction is a common trick. Omitting the first clauses would mean that all propositional variables would be false, by default.

In the first definition of semantics in the example, the actual semantics are really hidden in the clauses defining axioms, just as it would be in a Hilbert systems. In the second definition the semantics are expressed explicitly in the second set of clauses, which is more similar to a natural deduction system. Another advantage is that the second definition is deterministic, while the first is not.

While a monotone partial inductive definition defines sets that could be defined by Horn clause definitions, it does in general permit the derivation of more **clauses** (it has a larger **cover**).

DEFINITION 3.12 Cover of a partial inductive definition
The set of **clauses** $Cov(P) = \{C \mid \vdash_P C\}$, is the **cover** of the partial inductive definition P. ■

While Def(P) is the set defined by P, Cov(P) can be seen as the "logic" defined by P. Cov(P) relates also to the properties P has as a definition, not only to what it defines. Def(P)=Def(P′) need not imply Cov(P)=Cov(P′) as can be seen by the simple example $P=\{\Rightarrow a\}$, $P'=\{\Rightarrow a, b \Rightarrow b\}$.

In particular there are non-Horn clause definitions defining sets that could also be defined by Horn clause definitions, but where no Horn clause definition could give the same cover. A simple example is the liar paradox definition of example 3.9, which defines the set {a}. This set is also defined by the Horn clause definition $\{\Rightarrow a\}$. The clause $a \Rightarrow b$, on the other hand, which is in the cover of the paradoxical definition could never be derived using a Horn clause definition defining the same set.

When we dealt with Horn clause definitions, we introduced a "model-theoretic" relation $\models$, in addition to the "proof-theoretic" relation $\vdash$. We will show briefly how this can be done for the full calculi of partial inductive definitions.

DEFINITION 3.13 Partial consequence relations
A **partial consequence relation** is a relation for which the $\vdash\Rightarrow$ and $\Rightarrow\vdash$ inference rules would give valid inferences, were the relation to take the place of $\vdash$ in these rules. ∎

The "partiality" is here used to contrast this concept from the usual concept of consequence relation, which should be transitive. Transitivity may hold, but is not required of our consequence relations, as we have seen.

DEFINITION 3.14 P-closed relations
A **P-closed relation** is a relation for which the $\vdash$P and P$\vdash$ inference rules would give valid inferences in the calculus for the partial inductive definition P, were the relation to take the place of $\vdash$ in these rules. ∎

DEFINITION 3.15 $\models$ for partial inductive definitions
The relation $\models$ for partial inductive definitions is the smallest P-closed partial consequence relation that includes F,e $\models$ e, for all F and e. ∎

Just as in the case of Horn clause definitions, $\vdash$ and $\models$ are equivalent. We will not show the proof here, the interested reader is referred to [6]. We will continue the practise of using $\vdash$ when we refer specifically to derivability and $\models$ when we are mainly interested in something being defined. $\models$ F, where F is a set of clauses, will be a shorthand for "$\models$ C, for all C$\in$ F", in the same way as for $\vdash$.

As mentioned earlier, some classes of partial inductive definitions have "logiclike" properties. We will now characterize these properties in more detail. Unfortunately, we cannot give any general criteria for which definitions have these properties. This is something that must be shown separately for each definition.

Earlier we said that assuming an atom that was not defined by any clause of the definition was a "contradictory" assumption. Clearly, it might be that $\vdash$ e does not hold even if there is a clause E $\Rightarrow$ e in the definition (because E does not hold). If e has a "contradictory" behavior when used as an assumption, we will say that e is false. If every atom either holds or is false, the definition is called complete.

DEFINITION 3.16 Falsity

Given a definition P and some E (a clause or a set of clauses). If $E \models_P C$ holds for all clauses or atoms C, then E is **false** in P. ∎

DEFINITION 3.17 Completeness of a partial inductive definition

Given a partial inductive definition P. If for all **atoms** $e \in U$, either $\models_P e$ holds or e is false in P, then P is **complete.** ∎

If this completeness property for atoms holds it also holds for arbitrary clauses and sets of clauses as the next two theorems show.

THEOREM 3.18 Completeness with respect to arbitrary clauses

If a partial inductive definition P is complete, then $\models_P C$ holds, or C is false in P, for all clauses C.

PROOF

By induction on C:

If C is an atom, by definition.

Otherwise C is $E \Rightarrow e$: Assume $\vdash_P e$. We have $E \vdash_P e$ by monotonicity, and so $\vdash_P E \Rightarrow e$. Assume instead that $\nvdash_P e$, then e must be false in P, that is $e \vdash_P D$ holds for all clauses D. We have two cases:

- For all $C' \in E$, $\vdash_P C'$ holds: By the $\Rightarrow \vdash$ rule, $E \Rightarrow e \vdash_P D$, i.e. $C \vdash_P D$, so C is false in P.
- For some $C' \in E$, $\vdash_P C'$ does not hold: By the induction hypothesis then, C' is false in P. So, in particular, $C' \vdash_P e$. $E \vdash_P e$ follows by monotonicity, and so $\vdash_P E \Rightarrow e$, i.e. $\vdash_P C$.

Thus in all cases either $\models_P C$, or C is false in P. ∎

THEOREM 3.19 Completeness with respect to sets of clauses

If a partial inductive definition P is complete, then $\models_P F$ holds, or F is false in P, for all sets of clauses F.

PROOF

Assume that $\models_P F$ does not hold. Then there exists at least one clause $C \in F$ such that $\not\models_P C$. By completeness for clauses, $C \models_P G$ for any G, and, by monotonicity, $F \models_P G$, i.e. F is false in P.

Thus $\models_P F$ holds or F is false in P. ∎

The relation $\models$ is not transitive in general. We will say that a definition P is total, if the relation $\models$ is transitive. Since the relation $\vdash$ is the same as $\models$, this means that the cut rule holds in the calculus based on P.

DEFINITION 3.20 Totality

If, for all sets of clauses or atomsG, $F \models_P G$ and $G \models_P C$ implies $F \models_P C$, then P is a **total** definition. ∎

The reason partial inductive definitions are called "partial" is that they are, in general, not total.

If the partial inductive definition is both total and complete, then the symbol $\Rightarrow$ can in fact be read as logical implication. Expressed another way, the symbol $\models$ can be read as "if ... then".

THEOREM 3.21 If-then reading of $\models$ under certain conditions
$F \models_P G$ is equivalent to "if $\models_P F$ then $\models_P G$", iff P is total and complete.
PROOF
Only if-part:
Assume that $F \models_P G$ is equivalent to "if $\models_P F$ then $\models_P G$".
For P to be complete, either $\models_P F$ or $F \models_P G$, or in other words: not $\models_P F$ implies $F \models_P G$. Assume not $\models_P F$. Then "if $\models_P F$ then $\models_P G$" holds trivially. But this is equivalent to $F \models_P G$, so completeness is established.
To show that P is total, we assume $F \models_P H$ and $H \models_P G$ (for unspecified H). If not $F \models_P G$, then by the equivalence (with F and G), $\models_P F$ and not $\models_P G$. Using the equivalence again (with F and H), we have "if $\models_P F$ then $\models_P H$", and so $\models_P H$. Using the equivalence a third time (with H and G), we have "if $\models_P H$ then $\models_P G$", and so $\models_P G$. But this leads to a contradiction, so $F \models_P G$ must hold after all, i.e. P is total.
If-part:
Assume that P is total and complete.
To show the left-to-right direction of the equivalence assume $F \models_P G$ and $\models_P F$. By totality we immediately have $\models_P G$ and this direction is proved.
To show the right-to-left direction of the equivalence assume that if $\models_P F$ then $\models_P G$. If $\models_P F$ holds, then $\models_P G$ and consequently $F \models_P G$ by monotonicity of the calculi. Should $\models_P F$ not hold, then by completeness $F \models_P G$. In either case the right-to-left direction is proved. ∎

4. DEFINING LOGIC USING PARTIAL INDUCTIVE DEFINITIONS

This section constitutes a larger example which will also be used in the following sections.

We will define the syntax and semantics of the predicate calculus. The semantics will permit us to determine the truth value of any formula in a given model. In the simplest case, the notions "truth value" and "model" correspond to the standard notions, but they may also be extended. We will return to this point further on.

Defining the syntax is straightforward and follows exactly the pattern of logic textbooks: ($\leftrightarrow$ and $\neg$ are omitted from this definition, and will be regarded as abbreviations in the usual way.)

$\Rightarrow A$	for all A that are atomic formulae of the predicate calculus
$A,B \Rightarrow A \wedge B$	for all A,B
$A,B \Rightarrow A \vee B$	for all A,B
$A,B \Rightarrow A \rightarrow B$	for all A,B
$A \Rightarrow \forall x\ A$	for all A and all variables x
$A \Rightarrow \exists x\ A$	for all A and all variables x

Next, we give a definition of the semantics among the same lines as the second alternative of example 3.11.

$A,B \Rightarrow A \wedge B$	for all A,B
$A \Rightarrow A \vee B$	for all A,B
$B \Rightarrow A \vee B$	for all A,B
$(A \Rightarrow B) \Rightarrow A \rightarrow B$	for all A,B
$\{A[t/x]\}t \in TERM \Rightarrow \forall x\ A$	for all A
$A[t/x] \Rightarrow \exists x\ A$	for all A and all $t \in TERM$

where TERM is the set of objects of the intended interpretation (the Herbrand universe for instance).

Here $A \wedge B$ is defined to hold if both A and B holds, $A \vee B$ to hold if either A or B holds, and $A \rightarrow B$ to hold if B follows from A. Note the similarities between these clauses and the corresponding introduction rules of natural deduction. $\forall x\ A$ is defined to hold if A[t/x] holds for all terms t. $\exists x\ A$ is defined to hold if A[t/x] holds for some term t. These clauses do not correspond directly to the introduction rules of natural deduction. The reason is that in natural deduction we deal with **logical** consequence, meaning that the consequence should hold in any interpretation, while here we are concerned with the truth value in some intended interpretation.

Indeed, let us see what the $\vdash P$ and $P \vdash$ inference rules will look like when they are used with this definition. We will list all instances of the $\vdash P$ and $P \vdash$ rules when they are used with each set of clauses in the definition of the semantics. If the symbol $\Rightarrow$ should appear, we immediately use the $\vdash\Rightarrow$ or $\Rightarrow\vdash$ rule to get rid of it.

$$\frac{F \vdash A \quad F \vdash B}{F \vdash A \wedge B}\vdash P \qquad \frac{F,A,B \vdash C}{F,A \wedge B \vdash C}P\vdash$$

$$\frac{F \vdash A}{F \vdash A \vee B}\vdash P \quad \text{or} \quad \frac{F \vdash B}{F \vdash A \vee B}\vdash P \qquad \frac{F,A \vdash C \quad F,B \vdash C}{F,A \vee B \vdash C}P\vdash$$

$$\frac{\dfrac{F,A \vdash B}{F \vdash A \Rightarrow B}\vdash\Rightarrow}{F \vdash A \to B}\vdash P \qquad \frac{\dfrac{F \vdash A \quad F,B \vdash C}{F,A \Rightarrow B \vdash C}\Rightarrow\vdash}{F,A \to B \vdash C}P\vdash$$

$$\frac{\{F \vdash A[t/x]\}t \in TERM}{F \vdash \forall x\, A}\vdash P \qquad \frac{F,\{A[t/x] \mid t \in TERM\} \vdash C}{F,\forall x\, A \vdash C}P\vdash$$

$$\frac{F \vdash A[t/x]}{F \vdash \exists x\, A}\vdash P \text{ for all } t \in TERM \qquad \frac{\{F,A[t/x] \vdash C \mid t \in TERM\}}{F,\exists x\, A \vdash C}P\vdash$$

These instances are almost exactly the inference rules for the sequent calculus for the (intuitionistic) predicate logic! A minor difference is in the $P\vdash$-rules for $\wedge$ and $\forall$, where we get several new assumptions instead of a single one as in the sequent calculus. This difference turns out to be insignificant. As $\vdash$ is monotone, we can simply drop the extra assumptions (corresponding to the "weakening" structural rule of the sequent calculus). The major differences can be found in the $\vdash P$ rule for $\forall$ and the $P\vdash$ rule for $\exists$. In logic the corresponding inference rules are

$$\frac{F \vdash A[y/x]}{F \vdash \forall x\, A}\forall\text{-right} \qquad \frac{F,A[y/x] \vdash C}{F,\exists x\, A \vdash C}\exists\text{-left}$$

where y is an "eigenparameter". Since we have a given universe, we can infer that $\forall x\, A$ from the fact that $A[t/x]$ holds for all terms t – and similarly for $\exists$. It should be clear that the inference rules we obtain are valid for a given universe.

Since we have listed all instances of the $\vdash$P and P$\vdash$ rules and found each to be valid, we conclude that our partial inductive definition has faithfully modelled the semantics of predicate logic. As A$\rightarrow$B is defined to be A $\Rightarrow$ B the symbol $\Rightarrow$ can is this case be read as implication. Using theorem 3.21, we see that our definition is an example of a definition that is both total and complete.

To define predicate logic with equality, clauses defining equality inductively could be added.

$$x_1=y_1,\ldots,x_n=y_n \Rightarrow f(x_1,\ldots,x_n)=f(y_1,\ldots,y_n)$$ for all terms $x_1,\ldots,x_n,y_1,\ldots,y_n$ and function symbols f.

$x\neq y$ will be an abbreviation of $\neg$ x=y in the usual way.

Derivations using the definition of predicate logic will in general be infinite, as the quantifiers introduce an infinite number of cases. We will use a simple finite representation of these infinite derivations. Infinities are introduced by using the $\vdash$P rule with universal quantification and the P$\vdash$ rule with existential quantification. We will write down these applications as follows:

$$\frac{F\vdash A[x^*/x]}{F\vdash \forall x\, A}\vdash P \qquad\qquad \frac{F,A[x^*/x]\vdash C}{F,\exists x\, A\vdash C}P\vdash$$

The substitution of a variable for a starred variable is an entirely formal construction. The presence of a starred variable means that we actually have a number of different cases, each one obtained by replacing the starred variable by a concrete term. The actual set of entities over which the starred variable ranges is assumed to be the set of terms. If not, we make a note as to what the set is.

Infinities could also be introduced by the using the P$\vdash$ rule with universal quantification, but this can be avoided by immediately dropping all but one of the assumptions:

$$\frac{F,A[t/x]\vdash C}{F,\forall x\, A\vdash C}P\vdash \quad \text{for some } t\in TERM$$

In this case t could also be a starred variable, if the actual t would depend on which of several cases we were in.

To split cases, we use a "pseudo inference rule", which is not an inference rule at all, but which serves the purpose of splitting a derivation into cases, where a starred variable represents different concrete terms.

EXAMPLE 4.1

We can now write down an infinite derivation such as:

$$\dfrac{\dfrac{\dfrac{}{\vdash \varnothing=\varnothing}\vdash P}{\vdash \varnothing=\varnothing \vee \varnothing\neq\varnothing}\vdash P \qquad \dfrac{\dfrac{\dfrac{}{x^*=\varnothing \vdash \bot}\ P\vdash}{x^*\neq\varnothing}\ \vdash\Rightarrow}{\vdash x^*\neq\varnothing}\ \text{[Where } x^*\neq\varnothing\text{]}}{\vdash x^*=\varnothing \vee x^*\neq\varnothing}\vdash P\ \text{[Where } x^*\neq\varnothing\text{]}}\ \text{case split}$$

($x^*=\varnothing$ cannot hold if $x^*\neq\varnothing$)

($x^*\neq\varnothing$ is an abbreviation)

$$\dfrac{\vdash x^*=\varnothing \vee x^*\neq\varnothing \qquad A(x^*)\vdash A(x^*)}{(x^*=\varnothing \vee x^*\neq\varnothing)\Rightarrow A(x^*)\vdash A(x^*)}\Rightarrow\vdash$$

$$\dfrac{(x^*=\varnothing \vee x^*\neq\varnothing)\Rightarrow A(x^*)\vdash A(x^*)}{(x^*=\varnothing \vee x^*\neq\varnothing)\rightarrow A(x^*)\vdash A(x^*)}\ P\vdash$$

$$\dfrac{(x^*=\varnothing \vee x^*\neq\varnothing)\rightarrow A(x^*)\vdash A(x^*)}{\forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\vdash A(x^*)}\ P\vdash$$

$$\dfrac{\forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\vdash A(x^*)}{\forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\vdash \forall x\,A(x)}\vdash P$$

$$\dfrac{\forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\vdash \forall x\,A(x)}{\vdash \forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\Rightarrow \forall x\,A(x)}\vdash\Rightarrow$$

$$\dfrac{\vdash \forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\Rightarrow \forall x\,A(x)}{\vdash \forall x\,((x=\varnothing \vee x\neq\varnothing)\rightarrow A(x))\rightarrow \forall x\,A(x)}\vdash P$$

In the case split, $x^*\neq\varnothing$ in the right part. In the left part, $x^*=\varnothing$, so $\varnothing$ is substituted for x^*. ■

We said in the beginning of this section that we could use the partial inductive definition of the semantics to find the truth value of a formula in a given model. How do we give the model? As the definition does not contain clauses defining any atomic formulae, they are all absurd by default. Thus the definition as it stands implies a model where every atomic formula is false.

EXAMPLE 4.2

We have $\vdash \neg A$ $\vdash B\rightarrow B$ $\vdash A\rightarrow B$ $\vdash (A\wedge B)\rightarrow \neg(\neg A \vee \neg B)$

but $\nvdash A$ $\nvdash \exists x\,R(x)$ ■

If we want a model where some atomic formulae are true, we will add clauses of the form

$\Rightarrow P$ for every atomic formulae P that we want to be true.

EXAMPLE 4.3

We add the clauses $\Rightarrow A$ and $\Rightarrow R(1)$, making A and R(1) true.

We now have $\vdash A$ $\vdash \exists x\,R(x)$ $\vdash B\rightarrow B$ $\vdash (A\wedge B)\rightarrow \neg(\neg A \vee \neg B)$

but $\nvdash \neg A$ $\nvdash A\rightarrow B$ ■

If we call the definition of the semantics T and the definition of true atomic formulae I, we get

$\vdash_{T,I} A$ iff the formula A is true in the model corresponding to the definition I.

So Def(I) are those atomic formulae that are true in the model, i.e. the model itself. All other atomic formulae are false.

These ideas can be used to define the semantics of any formal language where the concepts of a formula "holding" or not can be expressed by a partial inductive definition. Both this general idea and the particular semantics will be used frequently in the following sections.

In the simple way we have done it in this case, the language must have the property that the meaning of a formula is independent of the context in where it occurs. Languages where this is not true, e.g. modal logics, could still be handled by defining when a pair consisting of a formula and a context holds. However, we have not investigated this approach any further.

We can permit I to be an arbitrary partial inductive definition. The simple relationship between a model and an inductive definition will then not hold in general.

EXAMPLE 4.4
Consider $I=\{\neg a \Rightarrow a\}$. We then have $Def(I)=\{a\}$, but both $\vdash_{T,I} a$ and $\vdash_{T,I} \neg a$, i.e. both a and $\neg a$ are true, which is obviously not possible in any standard model (this is essentially the same example as example 3.9). ■

In such cases, we will say that I corresponds to a generalized model. If, however, I is total and complete, it will correspond precisely to the ordinary model Def(I) and so gives us the standard model concept. These ideas have been developed in [5].

To give an example of the use of generalized models, we will show a generalized model that can be used to show logical truth, as opposed to truth in a (standard) model. We will let I be the following "undefining" model, using the same trick as in example 3.11:

$A \Rightarrow A$ for all atomic formulae A

As this definition leaves undefined the truth or falsity of every atomic formula, we could expect it to make a formula true if and only if that formula was true in all cases, i.e. if the formula was a theorem.

For the propositional fragment of predicate logic, this model does indeed work as desired. For full first order logic, we have the additional complication that the $\forall$-right and $\exists$-left rules we obtain with our definition of the semantics for predicate logic work with a premise for each element in TERM, while these rules in the sequent calculus work with one premise containing an eigenparameter. This could permit us to infer formulae that are not actually therorems of predicate logic. However, if we let TERM contain an infinite number of constants, then our inference rules will be essentially equivalent to the standard ones, and the undefining model will give logical truth.

It should be noted that as we have only defined the truth of a formula constructively, using the truth of its subformulae, a formula can not hold by virtue of reductio ad absurdum. Thus, we can only derive theorems that are derivable in intuitionistic logic. If we wanted to derive theorems of classical logic, we would have to extend the definition of predicate logic with clauses of the form $\neg\neg A \Rightarrow A$.

EXAMPLE 4.5

With the "undefining" model, we have $\vdash (A \wedge B) \rightarrow \neg(\neg A \vee \neg B)$

but $\nvdash A \quad \nvdash \neg A \quad \nvdash A \vee \neg A$ ■

If the clauses defining equality were included, we would have $\vdash \forall x\ (x=\varnothing \vee x\neq\varnothing)$ even with the undefining model, as these clauses give an interpretation of equality.

Other generalized models have properties that lay between the "undefining" model and standard models. We mentioned in section 2 that Horn clauses in logic can be interpreted as Horn clause definitions. Thus a pure Prolog program could be though of a a generalized model - the ideas of the following sections are based on this.

5. SPECIFICATIONS

We will now consider the question of how we can express programs and their specifications using partial inductive definitions. We are mainly interested in specifying two things: data types and programs. A data type can be fully specified by giving the set of all objects of the type. A program can be specified in the traditional way by giving its input condition and input-output relation (these will be treated in more detail in a section 7). These are unary and binary relations, both of which can again be expressed as sets. Thus we are left with the fundamental problem of specifying sets.

The obvious way of specifying a set in the context of partial inductive definitions would be to simply give an inductive definition of the set. We want to avoid this as the language of clauses is not in general a very convenient specification language, and in any case it would be an advantage if different specification languages could be used depending on the particular application of the general framework of programming calculi. Furthermore, using partial inductive definitions as specifications would lead to methodological problems as the programs themselves, as we will see, are partial inductive definitions.

The approach we will take is to let the specification consist of a formula in some arbitrary specification language. The semantics of the specification language itself will be defined by a partial inductive definition in the same manner as we defined the semantics of the predicate calculus. The specification language could not be completely arbitrary, of course, as we must be able to define its semantics using a partial inductive definition. In the following examples, the predicate calculus and its definition from section 4 will be used as the specification language and its definition.

Let us now consider how the specification is used to specify a particular set. Assume that we use predicate logic as our specification language and that we have a formula Σ that is our specification. Σ will be true in some models and false in others. As we intend Σ to be true, to avoid contradictions the set of possible models must be limited to those that make Σ true. Assume further that we have a special predicate $S'(x)$. For a given model and term t, $S'(t)$ will be either true or false (or possibly unspecified, we will discuss this shortly). We now say that those t for which $S'(t)$ is true is the set specified by the **pair** Σ and S'. For this approach to work, exactly the same $S'(t)$:s must be true in each possible model. An example should make this clear:

EXAMPLE 5.1

Suppose that Σ is $\forall x\ (L(x) \leftrightarrow x=\varnothing \vee \exists u\ \exists l\ (x=u.l \wedge L(l)))$ and S' is L.

Now t is a list iff L(t) is true in each model that makes Σ true. From Σ we see that $L(\varnothing)$ must be true if Σ is, so $\varnothing$ is a list. Also, L(u.l) must be true (u.l is a list) for any u, if L(l) is true (l is a list). Thus, by induction over the set of terms, we see that L(t) is true in exactly those cases where t is a list. ■

EXAMPLE 5.2

Suppose that Σ is as in the previous example, but S' is A.

In this case, the pair (Σ,A) does not specify anything. For any t, A(t) could be either true or false without changing the validity of Σ. So here Σ does not constrain the set of possible models enough to make the set determined by A(t) unique. ■

It could also be that the model did not assign a truth value at all to $S'(t)$, and that Σ were still true. This also means that the validity of Σ is not dependent on the truth or falsity of $S'(t)$, Again we have the situation that $S'(t)$ would not have a unique truth value, so we will not get a specification in this case either.

Here we have intuitively assumed that we know how to substitute t for x in A, getting A(t) from A(x). Different specification languages might have entirely different substitution properties, so we will need a general mechanism to model substitutions. What we really want is a way of mapping a term to an atomic formula, so that we see if this atomic formula is part of the model. For this purpose we will introduce the notion of formula functions. As the elements of the model are atomic formulae, a formula function maps a term (or a tuple of terms) into an atomic formula. Regarding S' as a formula function in the first example above, we would have

$$S' = \lambda t \,.\, L(x)[t/x]$$

So we see that substitutions are hidden inside the formula functions and need not concern us further.

Specification languages might not have the concepts "term" and "atomic formula". Regardless of the specification language used, however, we will keep the name "term" for such objects that belong to the sets we specify and "atomic formulae" for whatever makes up the elements of models.

As mentioned, this programming calculus is a general framework that can be applied to different specification languages. We have introduced several "parameters" that can be changed between different applications of the framework. Before we state the general definition of a specification we will list these parameters and what we require of them.

In the sequel, it is assumed that

- We have some **universe** U of "expressions". The elements of U are the atoms in the sense of the theory of partial inductive definitions. We make no requirements on U, except that it be large enough to include all expressions we need.
- We have some **specification language** $L \subseteq U$.
- The **semantics** of L is given by some partial inductive definition T (over U).
- We have some set of **atomic formulae** $B_L \subseteq U$ used to form models. The name B_L is chosen because, in the predicate calculus case, this set could be the Herbrand base.
- We have a set of **terms** H_L. The name H_L is chosen because, in the predicate calculus case, this set could be the Herbrand universe. What these terms actually are in general depends on the informal substitution properties of the specification language. H_L must be given inductively, so that we can do induction over it.
- We have a set, F, of possible **formula functions** from tuples of terms to B_L. For each formula function, the number of terms of the tuples is called its **arity**.
- We have some class R of partial inductive definitions over B_L, used to form **programs** (we will return to this in section 6).

So for each application of the general framework, we must decide upon U, L, T, B_L, H_L, F and R. In our **examples**, we will assume that

- L is the set of all formulae of first-order predicate logic.
- U is at least as large as L.
- B_L is the Herbrand base of the formulae involved.
- H_L is the Herbrand universe of the formulae involved.
- F are functions of the form $\lambda t_1,\ldots,t_n \,.\, P[t_1/x_1,\ldots,t_n/x_n]$, modelling first order substitution.
- T is the definition of the semantics of predicate logic from section 4.
- R is the set of Horn clause definitions, corresponding to pure Prolog programs

We can now state the definition of a specification

DEFINITION 5.3 Specifications

A pair (Σ,S') where $\Sigma \in L$ and S' is a formula function of arity 1, is **a specification of a set S** iff $\forall P \in R\ (\models_{P,T} \Sigma \rightarrow S=\{\xi \mid \models_P S'(\xi)\})$. ξ ranges over H_L, the set of terms. ■

Every model that makes the specification true includes atomic formulae of the form $S'(\xi)$ for exactly those ξ that are in S. When we want to specify sets of tuples (e.g. relations), this definition generalizes in the obvious way with formula functions of higher arities. The reason only definitions taken from the set R are considered as models will be explained in section 7.

EXAMPLE 5.4

To specify a program that finds the last element of a list, we specify its input-output relation. Such a specification can be the pair (Σ,LAST), where

Σ is $\forall a\, \forall z\, (last(a,z) \leftrightarrow (a=\varnothing \wedge z=\varnothing \vee \exists x\, \exists y\, (a=x.y \wedge (y=\varnothing \wedge x=z \vee y\neq\varnothing \wedge last(y,z)))))$

and $LAST(x,y) = last(x,y)$

The empty list does not have any elements. To permit the program to compute something for any list, we specify (arbitrarily) that the last element of the empty list is the empty list.

We now have that the input-output relation among other things holds between

$\varnothing$ and $\varnothing$, a.b.$\varnothing$ and b, 1.2.3.4.5.$\varnothing$ and 5. ■

6. PROGRAMS AND COMPUTATIONS

We want to keep the paradigm introduced by logic programming of regarding computations as deductions. A computation from a logic program, P, can be seen as the process of finding a proof of $P \vdash G$ (in **logic**, so $\vdash$ denotes derivability in logic for the next few paragraphs), where G is a formula derived from the given goal. Typically, G would be the goal with all variables existentially quantified and witnesses for these existentially quantified variables would provide the result of the computation (in Prolog systems, this is done by binding the variables). So if we had the (somewhat silly) Prolog program

```
foo(X.Y,X)
foo(a(X),b(Y)) :- foo(a(X),b(Y))
```

and executed the goal foo(a.b,U), this would correspond to finding a constructive proof of

$$\forall x\ \forall y\ foo(x.y,x),\ \forall x\ \forall y\ (foo(a(x),b(y)) \rightarrow foo(a(x),b(y))) \vdash \exists u\ foo(a.b,u)$$

which would indeed give a as a witness for u, as would be expected.

This raises the question of what the "program" really is, as the particular computation is determined not only by the program clauses, but also by the goal. Each different goal gives a different computation. The usual view in logic programming is that the program clauses comprise the program, and that the goal determines how the program is to be used [13]. In this way programs are naturally "multi-way" in that different arguments to the relation could be provided (be inputs) and different arguments could be searched for (be outputs) at different times.

Since we are interested in the correctness properties of a program, this view has the drawback that we would have to establish correctness of a program without knowing in which way the program would be used. The termination properties of the same program could vary widely depending on the form of the goal. It would certainly be possible to show termination for all possible forms of the goal, but this might be too a strong a requirement, if we were only interested in using the program in a certain way. The program above will not terminate in arbitrary cases (e.g. with the goal foo(a(X),Y)), but if we were only interested in cases were the first argument was a pair, termination would hold.

To remedy this difficulty, we will regard a program as a pair consisting of the program clauses and a goal (this is similar to the view taken by Hogger [11]). Having no inputs, such a program would not be very interesting, since it would compute the same results every time. To permit input we will define a program to be a pair consisting of the program clauses and a "prototype" goal - a function from the

input to a goal. If, given the program above, we wanted to compute the second argument of the relation given the first argument, the prototype goal would be $\lambda x.foo(x,U)$. So if we wanted the program to compute something from the input "a.b", we would apply the prototype goal to a.b, getting foo(a.b,U) to which we would add an existential quantifier and use in the manner shown above.

Essentially, the program defines a function from input to output, which has the advantage that traditional correctness concepts such as partial correctness and termination are immediately applicable.

As we want programs and computations to be expressed using the theory of partial inductive definitions, this discussion has been intended only to convey the ideas behind our definitions of programs and computations. When we look for a suitable way to express programs, the similarity between clauses of logic programs and inductive definitions suggests that the partial inductive definitions themselves could be used as programs. Execution of a program would then be equivalent to making a derivation in the calculus based on that program.

The major problem here is that there must exist efficient mechanisms for making these derivations. For Horn clauses such an efficient mechanism exists - SLD-resolution [13], which form the base of implementations of Prolog. As Horn clause definitions has great structural similarity to Horn clauses in logic, we should expect that there are just as efficient mechanisms in that case. Indeed, Horn clause definitions that are represented using first-order substitution are equivalent to Horn clauses in logic and SLD-resolution would work for them as well.

As further research in logic programming has shown, substantial extensions to Horn clauses can be made while keeping most of this efficiency. Thus we can expect that full partial inductive definitions could also be executed with reasonable efficiency. Work along the lines of providing computation mechanisms for Horn clause definitions and partial inductive definitions in general has been done by Aronsson [2] and Hallnäs [7].

In conclusion, when we have a program expressed as a partial inductive definition P, executing the program amounts to finding a derivation $\vdash_P G$, where G is again derived from the goal. As we do not have quantification in the calculi of partial inductive definitions, we must let the "prototype" goal be a function of two arguments - input and output - and move the existential quantification from the formula into the informal level, saying that if there exists a ζ such that $\vdash_P G(\alpha, \zeta)$, then ζ has been computed from α by the program (P,G). We can now state the exact definition of a program and a computation.

DEFINITION 6.1 Programs

A pair (P,G′) where P∈R and G′ is a formula function of arity 2, is **a program.** ■

Note the restriction that the definition part of a program must belong to R. For each application of the framework, R should be chosen so that there are efficient proof procedures for executing the programs.

DEFINITION 6.2 Computations

A program (P,G′) **computes** ζ **from** α, iff $\models_P G'(\alpha,\zeta)$. The actual process of finding a ζ for a given α such that $\models_P G'(\alpha,\zeta)$, is a **computation** of ζ from α. Thus the program defines the function $\lambda\,\alpha.$(a ζ such that $\models_P G'(\alpha,\zeta)$). The mechanism that is used to perform the computations is called a **proof procedure.** ■

If the program has more (or less) than one input and/or more than one output, α and/or ζ are tuples. It would also be possible to generalize the definition in the obvious way to a different number of arguments.

In case several different ζ:s can be found during the computation, the program becomes indeterministic. It is assumed that the proof procedure will choose some possible value for ζ and that the programmer does not care which one is chosen.

EXAMPLE 6.3

A program (P,APP) to append two lists:

P is	$\Rightarrow$ append(∅,x,x)	for all x,
	append(a,b,c) $\Rightarrow$ append(x.a,b,x.c)	for all x,a,b,c
and	APP((x_1,x_2),y)=append(x_1,x_2,y)	

so to append a.∅ and b.∅, we attempt to find a ζ such that $\models_P$ APP((a.∅,b.∅),ζ).
We find that a.b.∅ is such a ζ, since APP((a.∅,b.∅),a.b.∅)=append(a.∅,b.∅,a.b.∅) and $\models_P$ append(a.∅,b.∅,a.b.∅) ■

EXAMPLE 6.4

A program (P,DIFF) to compute the difference between two lists.

P is	$\Rightarrow$ append(∅,x,x)	for all x,
	append(a,b,c) $\Rightarrow$ append(x.a,b,x.c)	for all x,a,b,c
and	DIFF((x_1,x_2),y)=append(x_2,y,x_1)	

To find the difference between a.b.∅ and b.∅, we attempt to find a ζ such that $\models_P$ DIFF((a.b.∅,b.∅), ζ). Such a ζ is a.∅. From this example and the previous one, we see that the "multiway" properties of logic programs are easily accessible, simply by changing the formula function. ■

EXAMPLE 6.5

A program (Q,LAST) to find the last element of a list. If the list does not contain any elements at all, we compute the arbitrary value ∅.

Q is		⇒ last(∅,∅)	
		⇒ last(x.∅,x)	for all x,
	last(u.v,z)	⇒ last(x.u.v,z)	for all x,u,v,z
and	LAST(x,y)=last(x,y)		

To find the last element of a.b.∅, we attempt to find a ζ such that $\models_Q$ LAST(a.b.∅,ζ). Such a ζ is b. ■

7. CORRECTNESS PROPERTIES

Our view of a program as defining a function permits us to use traditional notions of program correctness [14], which we will briefly recapitulate. The correctness of a program is divided into two parts: partial correctness and termination.

Partial correctness means that the program can never compute anything that is incorrect. To specify what is correct or not, a relation between input and valid output (the input-output relation) is associated with the program. Even if the program were partially correct, there would be no guarantee that it did compute output in all cases. Instead, the program might loop or fail in some way. A program in our framework with an empty definition part, for instance, would always be partially correct, regardless of what we wanted the program to compute. As such a program does not compute anything at all, it obviously cannot compute anything incorrect, and so it would be trivially partially correct.

To ensure that the program does compute something, the termination property must be shown. Programs with this property always yield an output for every input. Typically, we would not be interested in termination for all conceivable input. A program to perform some computation on lists could hardly be expected to compute anything given a number as input. To take care of this, we determine an input condition, which holds for all input that we are interested in. We then say that the program terminates if it yields an output for every input fulfilling this input condition.

As partial correctness or termination alone are not enough, we must show that the program has both these properties. Such a program is said to be totally correct. Total correctness is always with respect to a certain input-output relation and a certain input condition.

We will now formalize these notions within the framework of our programming calculus:

DEFINITION 7.1 Input-output relations
A binary relation G is an **input-output** relation of some program if $G(\alpha,\zeta)$ holds in precisely those cases where ζ is a valid output from the program, given the input α. ■

DEFINITION 7.2 Input conditions
A unary relation A is an **input condition** of some program, if $A(\alpha)$ holds for those input for which we are interested in computing an output. ■

Definitions 7.1 and 7.2 (and subsequent definitions depending on them) generalize in the obvious way to programs with more than one output and/or more than one (or no) input parameters.

DEFINITION 7.3 Partial correctness

A program (P,G′) with input-output relation G is **partially correct with respect to** G iff $\forall\alpha\forall\zeta$ ($\models_P$ G′(α,ζ) $\rightarrow$ (α,ζ)$\in$ G). (For every input, every output that can be computed is permitted by the input-output relation). ∎

Termination adds a complication for us. We considered an execution to be a process where a certain derivation is found. If we fail to find such a derivation, the program does not terminate. This can happen for two different reasons. In the simple case there just is no such derivation, so none can be found. In the other case there is indeed such a derivation, but our proof procedure is unable to find it. The first of these cases is easy to characterize formally. The second one is not, as it involves the operational properties of the proof procedure, something that is not characterized within our framework.

Although this is certainly a weakness of this approach, it is a weakness that is shared with the "deductive" approaches to program correctness for logic programming [3] [8] [10]. There are a great number of proof procedures possible, each with different properties. To go into their operational properties is beyond the scope of this work. We will simply assume that any proof procedure used with our framework is complete, i.e. that it can find a derivation if there is one to be found, at least for those derivations that correspond to executions. Indeed, this is an important point to consider when we choose the parameters (especially R and F) for a particular application of our general framework. For the important case where the definition part of programs (the elements of R) are Horn clause definitions and F is the set of functions modelling first order substitution, we have exactly Horn clauses in first-order logic. For them breadth-first SLD-resolution [13] is a complete proof procedure.

When we talk about the proof procedure independent part of termination, we call a program complete if there exists a derivation of an output for any input fulfilling the input condition. So when the program is complete there is always a derivation for the proof procedure to find, and if the proof procedure is able to find it, the program terminates.

Note that if no derivation exists, it is quite possible for the proof procedure to discover that fact and terminate, reporting "failure" to find a derivation. In this case the **program** itself does not terminate, as no output was produced. Instead it is said to fail.

DEFINITION 7.4 Completeness of programs

A program (P,G′) **is complete** with respect to input condition A, where A is a unary relation (set) over H_L iff $\forall\alpha\in$ A $\exists\zeta \models_P$ G′(α,ζ). (For every input, there is some output that can be computed.) ∎

DEFINITION 7.5 Termination

A program (P,G') with input condition A **terminates** iff, for all $\alpha \in A$, the proof procedure used to perform computations is able to find a ζ that is computed from α by the program. ∎

THEOREM 7.6 Criterion for termination

A program (P,G') with input condition A **terminates** if it is complete, and the proof procedure used for execution is complete.

PROOF

If the proof procedure is complete, then a derivation $\models_P G'(\alpha,\zeta)$, can always be found if one exists. Since the program is complete, one always does exist for every $\alpha \in A$, thus a ζ can always be found, so the program terminates. ∎

DEFINITION 7.7 Total correctness

A program (P,G') with input-output relation G and input condition A is **totally correct** iff it is both partially correct with respect to G and terminates with respect to A. ∎

Let us now consider how we can give formal criteria for the partial correctness and completeness of a program. The way we have defined specifications, a pair (Σ,G') is a specification of an input-output relation G if, in all models which make Σ true, $G'(\alpha,\zeta)$ holds for all (α,ζ) tuples in G. Suppose we could in some way make a connection between a program and a model such that the model contained all pairs (α,ζ) where ζ is the output computed by the program from the input α. The program would then fulfill the specification if and only if Σ was true in that model.

If Σ was not true, then it could be that the model was too large, i.e. that some $G'(\alpha,\zeta)$ was in the model, but (α,ζ) was not in the input-output relation G. In that case the program would have computed a ζ that was not permitted by the input-output relation, i.e. the program would not be partially correct. On the other hand, it could be that the model was too small, i.e. that some $G'(\alpha,\zeta)$ was missing from the model. In that case, the program would be incomplete, as it would never be able to compute that ζ from the given α.

It turns out that it is possible to make a very simple connection between the program and a model. Suppose that the specification of the input-output relation is (Σ,G') and that the program is (P,G''). If the formula functions G' and G'' are the same, the desired model is simply the set inductively defined by P!

Let us see how this can be. Def(P) is the set of all e such that $\models_P e$. If (α,ζ) is some pair of input and corresponding output from (P,G''), then we must have $\models_P G''(\alpha,\zeta)$, in other words $G''(\alpha,\zeta) \in$ Def(P). Now, since $G'=G''$, Def(P) contains all $G'(\alpha,\zeta)$, which was the requirement on the model.

That programs can be used to generate models is the reason why the possible models in definition 5.3 were taken from the set R.

EXAMPLE 7.8

Consider the specification (Σ,A) of the program (P,A) where

Σ is $\quad \forall x\, \forall y\, (a(x,y) \leftrightarrow y=1)$

P is $\quad \Rightarrow a(x,1) \qquad$ for all x

and $\quad A(x,y) = a(x,y)$

This program is obviously partially correct, since it computes 1 given any input, which is what the input-output relation specified by (Σ,A) says it should. Def(P) is the set of all atomic formulae of the form a(x,1), for all x. This set is a model for Σ. ∎

It seems that a condition for partial correctness and completeness would be that Σ is true in the model generated by P, i.e. that $\models_{P,T} \Sigma$. This is a very simple condition, but unfortunately it does not always hold for a correct program. For one thing, it might be that for Σ to be true, the model must include atomic formulae that can not be generated by the program. This situation is quite possible (even likely) in practise, as a specification can include concepts that are used only by the programmer to help him structure the specification, but that do not occur in the program. To remedy this, we permit P to be extended to a new partial inductive definition P′, with some clauses that give us the additional atomic formulae required in the model. Of course, this extension must be made carefully, so that we do not introduce into Def(P) an atomic formula $G'(\alpha,\zeta)$ that either is an invalid input-output pair (making the program look partially incorrect even if it is not) or is a valid input-output pair that P alone could not give (hiding the fact that the program was incomplete).

EXAMPLE 7.9

Consider again the program (P,A) above, but with the specification (Σ',A) where

Σ' is $\quad \forall x\, \forall y\, (a(x,y) \leftrightarrow b(x,y)) \wedge \forall x\, \forall y\, (b(x,y) \leftrightarrow y=1)$

P is $\quad \Rightarrow a(x,1) \qquad$ for all x

and $\quad A(x,y) = a(x,y)$

(P,A) is partially correct with respect to this specification as well, since (Σ',A) specifies the same input-output relation, only in a more roundabout way. This time we have $\models_{P,T} \Sigma'$, however, since now b(x,1) must hold for all x as well, but does not. By extending P to P′ with the set of clauses {⇒ b(x,1), for all x}, we are able to show $\models_{P',T} \Sigma'$. P′ is obviously a monotone and conservative extension since exactly the same atomic formulae of the form a(x,y) are derivable in P′ as in P. ∎

Another difficulty is that if Def(P) contained other atomic formulae apart from the $G'(\alpha,\zeta)$, their presence in the model could make Σ false. If this happens the situation can not in general be remedied by extending P, as the first difficulty could. Fortunately, this possibility seems unlikely in practise, since it would mean that these extra atomic formulae would have a different (intuitive) meaning in the program and in the specification. If they did not, their presence would not invalidate Σ.

EXAMPLE 7.10

Consider again the specification (Σ',A) with the program (Q,A) where

Σ' is $\quad \forall x\, \forall y\, (a(x,y) \leftrightarrow b(x,y)) \wedge \forall x\, \forall y\, (b(x,y) \leftrightarrow y=1)$

Q is $\quad \Rightarrow a(x,1) \quad$ for all x

$\quad \Rightarrow b(x,2) \quad$ for all x

and $\quad A(x,y) = a(x,y)$

The program (Q,A) is also partially correct since it computes exactly the same thing as the programs in the previous examples did. Again, Σ' does not hold, since b(x,1) does not hold for all x. In this case, however, extending Q does not help, as we now have atoms of the form b(x,2) in Def(Q). These atoms must not be in any model that makes Σ' hold, and since we cannot get rid of them by extending Q, we cannot show partial correctness along the lines shown. ∎

This discussion is formalized by the following theorems which give sufficient criteria for program correctness. In the sequel we will use dom(f) and ran(f) to denote the domain and range of a function, respectively. f can also be a relation in general, in which case dom and ran will denote the straightforward generalization.

THEOREM 7.11 Condition for partial correctness

Given a program (P,G') with input-output relation G, specified by (Σ,G'), the program is **partially correct** if there exists an extension $P' \in R$ of P, monotone with respect to ran(G'), such that $\models_{P',T} \Sigma$.

PROOF

Since $\models_{P',T} \Sigma$, the definition of specifications gives $G=\{(\alpha,\zeta) \mid \models_{P'} G'(\alpha,\zeta)\}$, i.e.
$(\alpha,\zeta) \in G \leftrightarrow \models_{P'} G'(\alpha,\zeta)$. In particular $\models_{P'} G'(\alpha,\zeta) \rightarrow (\alpha,\zeta) \in G$. Since P' is a monotone extension we have $\models_{P} G'(\alpha,\zeta) \rightarrow \models_{P'} G'(\alpha,\zeta)$. Combining these we obtain $\models_{P} G'(\alpha,\zeta) \rightarrow (\alpha,\zeta) \in G$, for any α and ζ, which, by definition, gives partial correctness. ∎

THEOREM 7.12 Condition for completeness

Given a program (P,G′) with input condition A and input-output relation G, specified by (Γ,G′), the program is **complete** if A⊆dom(G) and there exists an extension P′∈R of P, conservative with respect to ran(G′), such that $\models_{P',T} \Sigma$. (The condition A⊆dom(G) is simply a requirement that the function the program computes is defined for all intended input).

PROOF

Since $\models_{P',T} \Sigma$, the definition of specifications gives G={(α,ζ) | $\models_{P'}$ G′(α,ζ)}, i.e. (α,ζ)∈G ↔ $\models_{P'}$ G′(α,ζ). For each α∈A, the condition A⊆dom(G) ensures that there exists some ζ such that (α,ζ)∈G. Using this we get $\models_{P'}$ G′(α,ζ). Since P′ is a conservative extension we know that $\models_{P'}$ G′(α,ζ) → $\models_{P}$ G′(α,ζ). Thus for every α∈A, there is some ζ such that $\models_{P}$ G′(α,ζ), which, by definition, gives completeness. ∎

THEOREM 7.13 Condition for total correctness

Given a program (P,G′) with input condition A and input-output relation G, specified by (Σ,G′) and executed by a complete proof procedure, the program is **totally correct** if A⊆dom(G) and there exists an extension P′∈R of P, monotone and conservative with respect to ran(G′), such that $\models_{P',T} \Sigma$.

PROOF

Immediate from theorems 7.6, 7.11 and 7.12. ∎

This theorem 7.13 is our main result and forms the basis for the methodology for verifying and synthesizing programs outlined in section 8. A complete example showing the application of it will also be given in that section.

To use this theorem, we must primarily be able to establish that $\models_{P',T} \Sigma$. This is done using the inference rules of the calculus for P′∪T. We must also show that the extension P′ of P is in fact both monotone and conservative with respect to ran(G′). This can be a difficult or even impossible task in general. For the important case of Horn clause definitions, however, we can give simple criteria for both monotonicity and conservativity. Monotonicity always holds, while conservativity can be ensured by imposing a simple syntactical restriction on how the extensions are formed. The following two lemmas show this.

LEMMA 7.14 Monotonicity of extensions to Horn clause definitions

If P is a Horn clause definition over U, then all extensions P′ of P are **monotone** with respect to any subset of U.

PROOF

Suppose that $\vdash_{P}$ e holds. Since P is a Horn clause definition, the derivation of e uses only ⊢P-rules. From the definition of the ⊢P-rule, it can be seen that every instance of the rule is still valid when P is extended with more clauses. Thus $\vdash_{P'}$ e. ∎

In the next lemma, we will use the notation atoms(P) to denote the set of all atoms occurring among the clauses of the partial inductive definition P. heads(P) will denote the set of clause heads in P.

LEMMA 7.15 Conservativity of extensions to Horn clause definitions
If P is a Horn clause definition, then an extension P′ of P is **conservative** with respect to a set S provided that $(S \cup atoms(P)) \cap heads(P' \setminus P) = \varnothing$.

PROOF
By induction over the depth of the derivation $\vdash_{P'} e$, for each $e \in (S \cup atoms(P))$. Suppose the lemma holds for derivations of depth less than the length of the given derivation. Since e is an atom the only inference rule possible in the final step of the derivation is a $\vdash$P-rule. Let the clause used by the rule be $E \Rightarrow e$. As $e \notin heads(P' \setminus P)$, the clause must belong to P. Then $E \subseteq atoms(P)$, so every premise of the rule has the form $\vdash_{P'} e'$, where $e' \in atoms(P)$, and so $e' \in (S \cup atoms(P))$. By the induction hypothesis, then, $\vdash_P e'$ holds for every e′. As $(E \Rightarrow e) \in P$, we have $\vdash_P e$. ∎

An extension could obviously be conservative under weaker conditions, but this condition has the advantage of being very simple to verify.

EXAMPLE 7.16
Suppose we have the following P, P′ and S:

$P = \{\Rightarrow a,\ b \Rightarrow c\}$
$P' = \{\Rightarrow a, \Rightarrow b,\ b \Rightarrow c\}$
$S = \{a, c\}$

P′ is clearly an extension to P, but the conditions of lemma 7.15 are not fulfilled, so it should not be conservative with respect to S. Indeed, we have $\nvdash_P c$, but $\vdash_{P'} c$. ∎

EXAMPLE 7.17
Suppose we instead have these P, P′ and S:

$P = \{\Rightarrow a,\ b \Rightarrow c\}$
$P' = \{\Rightarrow a,\ d \Rightarrow b,\ b \Rightarrow c\}$
$S = \{a, c\}$

P′ is still an extension to P, and the conditions of lemma 7.15 are not fulfilled. This time, however, it is a conservative extension, as both $\nvdash_P c$ and $\nvdash_{P'} c$. ∎

8. A METHODOLOGY FOR VERIFICATION AND SYNTHESIS

Having determined criteria for program correctness, we now outline a methodology for program verification and synthesis. Program verification involves showing that a particular program is totally correct with respect to a particular specification. Program synthesis involves the mechanical construction of a totally correct program from a particular specification.

Program verification is mainly a theorem proving task where a derivation of the specification formula is sought using the calculus given by the definition part of the program. Program synthesis involves both a program construction task and a simultaneous check that the program does not violate the specification. All of these tasks are search problems where it is essential that a good strategy exists. We will not discuss the search problems here, but simply outline the steps that need to be taken in each case and leave the search problems to further study.

Suppose we have an input-output relation G and input condition A and that G is specified by (Σ,G'). Formally, verifying that a program (P,G') is totally correct primarily involves finding an extension P', monotone and conservative with respect to $ran(G')$, such that $\models_{P',T} \Sigma$. We must also show that $A \subseteq dom(G)$, but this is a minor problem and we assume that it is done informally. We assume that the proof procedure used is complete.

Similarly, synthesizing a program involves finding a P' such that $\models_{P',T} \Sigma$ and then restricting P' to some P, keeping the usual monotonicity and conservativity constraints. This restriction could be arbitrary, with $P=P'$ as the trivial case, but as the additional information in P' is irrelevant to the correctness of the program, the program is in some sense "better", the smaller P is.

We can see that in both these cases the common problem is that of extending a partial inductive definition to an extension P' (keeping the usual constraints) such that $\models_{P',T} \Sigma$. In the verification case, the definition part of the program is extended, in the synthesis case an empty definition is extended. We will describe a procedure to do this extension. In the synthesis case, we must also conclude with a restriction. We cannot give a general procedure for this as it depends on what extensions are monotone and conservative for any particular set of programs, R.

To find the desired extension P', of a partial inductive definition P, we attempt to find a derivation of $\vdash_{P,T} \Sigma$. The derivation is performed in a bottom-up fashion, starting with Σ and applying inference rules backwards. For each inference rule that is applied, we get a set of premises which in their turn must be derived in the same manner. Whenever an axiom or inference rule without premises are used, the number of formulae remaining to be shown is reduced. If Σ holds, a complete derivation with no premises remain to be shown will eventually be obtained.

As the derivation process proceeds, we may find that at some point no inference rule is applicable. We may then extend P with a new clause to permit a backward inference to be made and the derivation to continue. P may be extended even if an inference would be possible otherwise, if this appears to lead to a more promising situation. Of course, P may only be extended in such a way that the extension still belongs to R.

In what cases could an extension be needed? The $\vdash\Rightarrow$ and $\Rightarrow\vdash$ rules do not use the definition at all, so they would not be affected by an extension. An extension before the application of a $P\vdash$ rule would be possible. Suppose we tried to show $F,e\vdash C$ using the $P\vdash$ rule. The form of this rule is

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

If the head of the new clause was not e, its inclusion would not affect the inference at all, so it could be postponed. If was e, the only effect would be to add a new premise to the inference, since there is one premise for every clause with e as its head. This would be counterproductive. Thus, the only case where an extension could be either necessary or useful would be an application of the $\vdash P$ rule.

Extending the definition is not always as simple as just adding a clause and applying a $\vdash P$ rule. Suppose that the clause $E\Rightarrow e$ is added. It might be that at some earlier point in the derivation we had an inference

$$\frac{F,E_1\vdash C \ldots F,E_n\vdash C}{F,e\vdash C}\,P\vdash$$

Where the clauses $E_1\Rightarrow e, \ldots, E_n\Rightarrow e$ all belonged to P.

If the clause $E\Rightarrow e$ is added to the definition, this inference would become invalid. The $P\vdash$ rule requires one premise for each clause in the database having e as its head. To make the inference valid again, we have to add the premise $F,E\vdash C$ to the set of premises remaining to be shown. To be able to do this we have to keep track of the conclusions of every occurrence of the $P\vdash$ rule. Thus an extension could result in a number of new premises to old inferences being generated.

Altogether the procedure works as follows. One of the premises remaining to be shown is selected. If we have just started, we have no premises yet, so Σ itself it taken. One of the following actions are then taken:

- The $\Rightarrow\vdash$ or $\vdash\Rightarrow$ rule is applied.
- The premise is an axiom, in which case no special action needs to be taken.
- The P$\vdash$ rule is applied. The selected premise, which becomes the conclusion of the inference is recorded.
- The $\vdash$P rule is applied.
- The definition is extended with a clause E $\Rightarrow$ e such that the extended definition still belongs to R, then the $\vdash$P rule is applied. For each recorded consequence F,e $\vdash$ C of earlier P $\vdash$ inferences, we add a new premise F,E $\vdash$ C.

If, at some point, none of these actions could be taken, we would back up to an earlier state of the derivation and make a different choice of action there. We could be forced to return to the initial state in this way. That would not prove that the program was incorrect, since our correctness criteria are sufficient, but not necessary.

EXAMPLE 8.1

To illustrate these principles we will give a complete example of the synthesis of a program.

We will synthesize a program to find the last element of a list. The specification (Σ,LAST) will be the same as in example 5.4, and the input condition A(x) will hold iff x is a list.

Σ is $\quad \forall a\, \forall z\, (last(a,z) \leftrightarrow (a= \varnothing \wedge z=\varnothing \vee \exists x\, \exists y\, (a = x.y \wedge (y=\varnothing \wedge x=z \vee y\neq\varnothing \wedge last(y,z)))))$

and $\quad LAST(x,y) = last(x,y)$

We begin by verifying that A$\subseteq$dom(LAST). As dom(LAST) is the entire Herbrand universe and A is the set of all list, this holds. Next, we attempt to find a P′ such that $\vdash_{P',T} \Sigma$. We do this using the procedure just described, starting with an empty definition and extending it to P′.

The derivation is presented in figures 8-1 to 8-4. Figure 8-1 shows the main part of the derivation. Figures 8-2 to 8-4 show derivations of added premises to P $\vdash$-rules when P′ is extended. The inference steps have been done in a bottom-up, left-to-right fashion. Steps of special interest are marked with numbers and commented upon below.

The Herbrand universe of this example consists of all terms constructed from the constant $\varnothing$ and the binary function symbol ".". When we do a case split, we have to consider all such terms.

Comments to figure 8-1:

1. last(a*,z*) $\leftrightarrow$ ... is an abbreviation for (last(a*,z*)$\rightarrow$...)$\wedge$(...$\rightarrow$last(a*,z*)).
2. Since P′ contains no clauses yet, the assumption last(a*,z*) is absurd in all cases. Since this is a P$\vdash$-inference, the conclusion of the step is recorded for possible later use.
3. Here we do the first case split. In the left part, we have the case where both a* and z* are $\varnothing$. In the right part we have all other cases.
4. Here we have to extend P′ to be able to derive $\varnothing=\varnothing \vdash$ last($\varnothing$,$\varnothing$). The simplest choice is the clause $\Rightarrow$ last($\varnothing$,$\varnothing$). We have one recorded conclusion of a P$\vdash$-rule, namely last(a*,z*) $\vdash$ ... from step 2. One case of this conclusion is last($\varnothing$,$\varnothing$) $\vdash$ ..., so the inference step with that conclusion now gets an extra premise which has to be derived. That derivation is shown in figure 8-2.
5. Since either a* or z* is different from $\varnothing$, one of the assumptions a*=$\varnothing$ or z*=$\varnothing$ will be absurd.
6. Since this is the case when a*$\neq$x*.y*, the assumption a*=x*.y* is absurd.
7. To save space, the uninteresting assumption x*.y*=x*.y* has been deleted from the premises.
8. Again, we have to extend P′, this time with the clauses $\Rightarrow$ last(x.$\varnothing$,x), for all x. Again, step 2 gets a new premise. The derivation of that premise is shown in figure 8-3.
9. In this case either y*$\neq\varnothing$ or x*$\neq$z*, so one of the assumptions y*=$\varnothing$ and x*=z* must be absurd.
10. P′ is extended again, with the clauses last(u.v,z) $\Rightarrow$ last(x.u.v,z), for all x,u,v,z. Step 2 gets a new premise. Its derivation is shown in figure 8-4.
11. y$\neq\varnothing$ is an abbreviation for y=$\varnothing\rightarrow\bot$.

This completes the derivation. During the process, we have added clauses, getting the following P′:

	$\Rightarrow$ last($\varnothing$,$\varnothing$)	
	$\Rightarrow$ last(x.$\varnothing$,x)	for all x
last(u.v,z)	$\Rightarrow$ last(x.u.v,z)	for all x,u,v,z

This definition cannot be restricted without violating the conservativity constraint, so (P′,LAST) is the desired program. It is guaranteed to be totally correct provided that the proof procedure used is complete. ∎

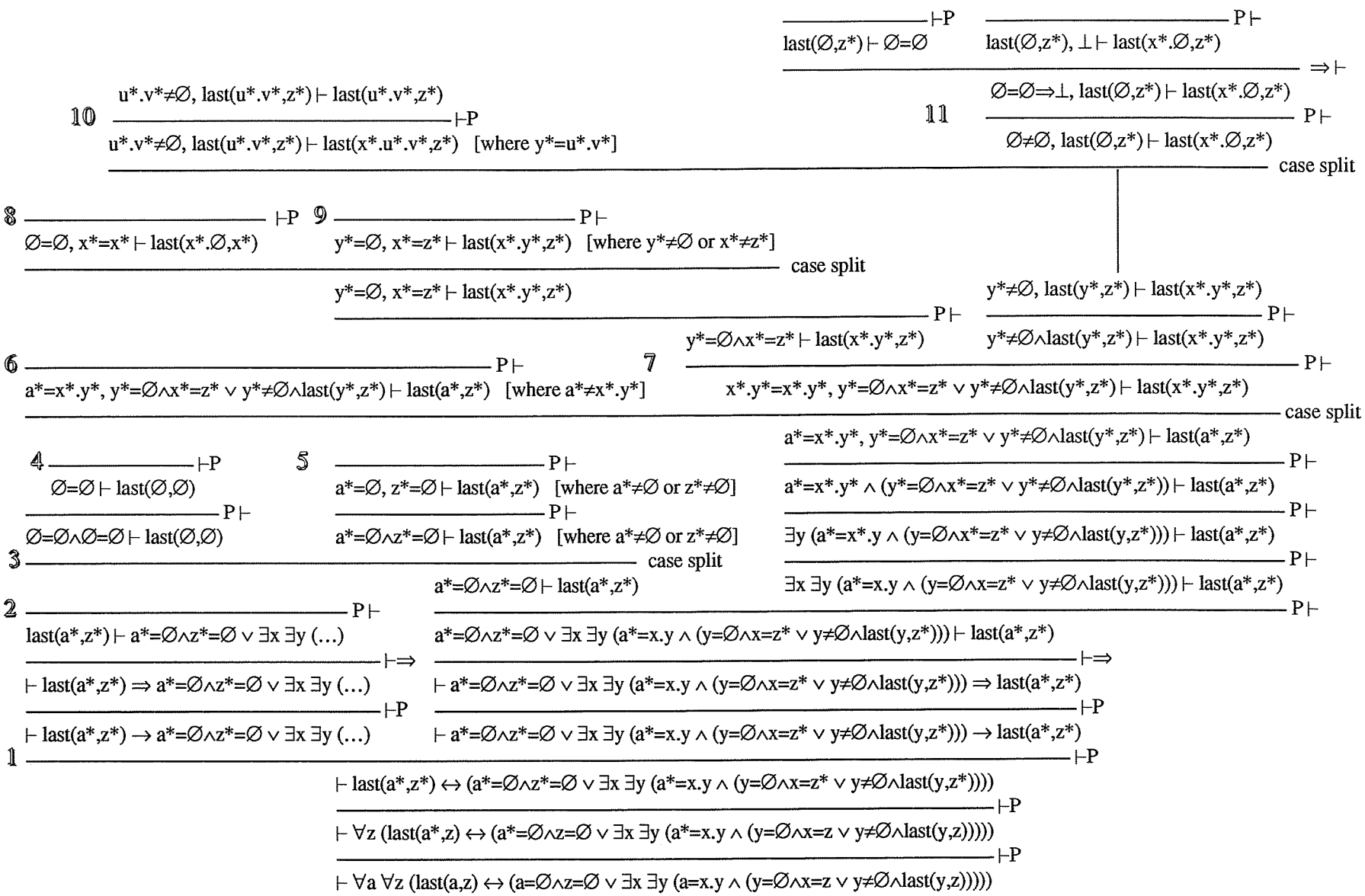
last(∅,z*) ⊢ ∅=∅ ⊢P
last(∅,z*), ⊥ ⊢ last(x*.∅,z*) P⊢
⇒⊢
10 u*.v*≠∅, last(u*.v*,z*) ⊢ last(u*.v*,z*) ⊢P
u*.v*≠∅, last(u*.v*,z*) ⊢ last(x*.u*.v*,z*) [where y*=u*.v*]
11 ∅=∅⇒⊥, last(∅,z*) ⊢ last(x*.∅,z*) P⊢
∅≠∅, last(∅,z*) ⊢ last(x*.∅,z*)
case split
8 ∅=∅, x*=x* ⊢ last(x*.∅,x*) ⊢P
9 y*=∅, x*=z* ⊢ last(x*.y*,z*) [where y*≠∅ or x*≠z*] P⊢
case split
y*=∅, x*=z* ⊢ last(x*.y*,z*) P⊢
y*≠∅, last(y*,z*) ⊢ last(x*.y*,z*) P⊢
y*=∅∧x*=z* ⊢ last(x*.y*,z*)
y*≠∅∧last(y*,z*) ⊢ last(x*.y*,z*) P⊢
6 a*=x*.y*, y*=∅∧x*=z* ∨ y*≠∅∧last(y*,z*) ⊢ last(a*,z*) [where a*≠x*.y*] P⊢
7 x*.y*=x*.y*, y*=∅∧x*=z* ∨ y*≠∅∧last(y*,z*) ⊢ last(x*.y*,z*) P⊢
case split
4 ∅=∅ ⊢ last(∅,∅) ⊢P
∅=∅∧∅=∅ ⊢ last(∅,∅) P⊢
5 a*=∅, z*=∅ ⊢ last(a*,z*) [where a*≠∅ or z*≠∅] P⊢
a*=∅∧z*=∅ ⊢ last(a*,z*) [where a*≠∅ or z*≠∅] P⊢
a*=x*.y*, y*=∅∧x*=z* ∨ y*≠∅∧last(y*,z*) ⊢ last(a*,z*) P⊢
a*=x*.y* ∧ (y*=∅∧x*=z* ∨ y*≠∅∧last(y*,z*)) ⊢ last(a*,z*) P⊢
∃y (a*=x*.y ∧ (y=∅∧x*=z* ∨ y≠∅∧last(y,z*))) ⊢ last(a*,z*) P⊢
3 case split
∃x ∃y (a*=x.y ∧ (y=∅∧x=z* ∨ y≠∅∧last(y,z*))) ⊢ last(a*,z*) P⊢
a*=∅∧z*=∅ ⊢ last(a*,z*)
2 last(a*,z*) ⊢ a*=∅∧z*=∅ ∨ ∃x ∃y (…) P⊢
a*=∅∧z*=∅ ∨ ∃x ∃y (a*=x.y ∧ (y=∅∧x=z* ∨ y≠∅∧last(y,z*))) ⊢ last(a*,z*) ⊢⇒
⊢ last(a*,z*) ⇒ a*=∅∧z*=∅ ∨ ∃x ∃y (…) ⊢⇒
⊢ a*=∅∧z*=∅ ∨ ∃x ∃y (a*=x.y ∧ (y=∅∧x=z* ∨ y≠∅∧last(y,z*))) ⇒ last(a*,z*)
⊢ last(a*,z*) → a*=∅∧z*=∅ ∨ ∃x ∃y (…) ⊢P
⊢ a*=∅∧z*=∅ ∨ ∃x ∃y (a*=x.y ∧ (y=∅∧x=z* ∨ y≠∅∧last(y,z*))) → last(a*,z*)
1 ⊢P
⊢ last(a*,z*) ↔ (a*=∅∧z*=∅ ∨ ∃x ∃y (a*=x.y ∧ (y=∅∧x=z* ∨ y≠∅∧last(y,z*)))) ⊢P
⊢ ∀z (last(a*,z) ↔ (a*=∅∧z=∅ ∨ ∃x ∃y (a*=x.y ∧ (y=∅∧x=z ∨ y≠∅∧last(y,z))))) ⊢P
⊢ ∀a ∀z (last(a,z) ↔ (a=∅∧z=∅ ∨ ∃x ∃y (a=x.y ∧ (y=∅∧x=z ∨ y≠∅∧last(y,z)))))

Figure 8-1

$$\dfrac{\dfrac{}{\vdash \varnothing=\varnothing}\vdash P \qquad \dfrac{}{\vdash \varnothing=\varnothing}\vdash P}{\vdash \varnothing=\varnothing\wedge\varnothing=\varnothing}\vdash P}{\vdash \varnothing=\varnothing\wedge\varnothing=\varnothing \vee \exists x\, \exists y\, (\varnothing=x.y \wedge (y=\varnothing\wedge x=\varnothing \vee y\neq\varnothing\wedge last(y,\varnothing)))}\vdash P$$

Figure 8-2

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \varnothing=\varnothing}\vdash P \qquad \dfrac{}{\vdash x^*=x^*}\vdash P}{\vdash \varnothing=\varnothing\wedge x^*=x^*}\vdash P}{\vdash x^*.\varnothing=x^*.\varnothing}\vdash P \qquad \dfrac{}{\vdash \varnothing=\varnothing\wedge x^*=x^* \vee \varnothing\neq\varnothing\wedge last(\varnothing,x^*)}\vdash P}{\vdash x^*.\varnothing=x^*.\varnothing \wedge (\varnothing=\varnothing\wedge x^*=x^* \vee \varnothing\neq\varnothing\wedge last(\varnothing,x^*))}\vdash P}{\vdash \exists y\, (x^*.\varnothing=x^*.y \wedge (y=\varnothing\wedge x^*=x^* \vee y\neq\varnothing\wedge last(y,x^*)))}\vdash P}{\vdash \exists x\, \exists y\, (x^*.\varnothing=x.y \wedge (y=\varnothing\wedge x=x^* \vee y\neq\varnothing\wedge last(y,x^*)))}\vdash P}{\vdash x^*.\varnothing=\varnothing\wedge x^*=\varnothing \vee \exists x\, \exists y\, (x^*.\varnothing=x.y \wedge (y=\varnothing\wedge x=x^* \vee y\neq\varnothing\wedge last(y,x^*)))}\vdash P$$

Figure 8-3

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{last(u^*.v^*,z^*),\, u^*.v^*=\varnothing \vdash \bot}P\vdash}{last(u^*.v^*,z^*) \vdash u^*.v^*\neq\varnothing}\vdash\Rightarrow \qquad last(u^*.v^*,z^*) \vdash last(u^*.v^*,z^*)}{last(u^*.v^*,z^*) \vdash u^*.v^*\neq\varnothing\wedge last(u^*.v^*,z^*)}\vdash P \qquad \dfrac{}{last(u^*.v^*,z^*) \vdash u^*.v^*=\varnothing\wedge x^*=z^* \vee u^*.v^*\neq\varnothing\wedge last(u^*.v^*,z^*)}\vdash P}{last(u^*.v^*,z^*) \vdash x^*.u^*.v^*=x^*.u^*.v^*}\vdash P \qquad \dfrac{}{last(u^*.v^*,z^*) \vdash x^*.u^*.v^*=x^*.u^*.v^* \wedge (u^*.v^*=\varnothing\wedge x^*=z^* \vee u^*.v^*\neq\varnothing\wedge last(u^*.v^*,z^*))}\vdash P}{last(u^*.v^*,z^*) \vdash \exists y\, (x^*.u^*.v^*=x^*.y \wedge (y=\varnothing\wedge x^*=z^* \vee y\neq\varnothing\wedge last(y,z^*)))}\vdash P}{last(u^*.v^*,z^*) \vdash \exists x\, \exists y\, (x^*.u^*.v^*=x.y \wedge (y=\varnothing\wedge x=z^* \vee y\neq\varnothing\wedge last(y,z^*)))}\vdash P}{last(u^*.v^*,z^*) \vdash x^*.u^*.v^*=\varnothing\wedge z^*=\varnothing \vee \exists x\, \exists y\, (x^*.u^*.v^*=x.y \wedge (y=\varnothing\wedge x=z^* \vee y\neq\varnothing\wedge last(y,z^*)))}\vdash P$$

Figure 8-4

9. CONCLUSIONS AND FURTHER DEVELOPMENT

We have presented a general framework for the specification, verification and synthesis of programs expressed using partial inductive definitions. The framework gives criteria for the correctness of programs and outlines a methodology for their verification and synthesis.

In comparison to other approaches to program verification and synthesis, our approach is most similar to the Logic Programming Calculus (LPC) of Hansson and Tärnlund et.al. [3] [8] and similar approaches [10]. Apart from the fact that programs are taken from different languages (which do have a large common subset), the major difference is that specifications and programs are different kinds of entities in the approach presented here. This has the advantage of permitting a greater generality in the choice of specification language. It does have the disadvantage that program transformations are not as easily done, as in the LPC. In the LPC specifications and programs are the same kind of entities, so the program to be transformed can be used as part of the specification for the transformed program [9], giving an elegant way of doing program transformations. It is not obvious how to do program transformations in the programming calculus based on partial inductive definitions. This should be investigated further.

Another difference is that in the LPC all reasoning required to establish correctness is done inside one formal system. Apart from the formal reasoning, we have done informal reasoning (e.g. to show that "$A \subseteq dom(G)$" in theorem 7.13), which could easier lead to mistakes, if care is not taken. On the other hand, this approach is one of the factors that makes the correctness criteria easy to establish, in contrast to the correctness criteria of the LPC, which have never been completely shown to be sufficient.

We have shown that it is possible to use this framework for the actual synthesis of concrete programs. As yet, we have not done any larger-scale experiments, but the similarities between our approach and that of the LPC, which has been tried on larger examples [4], suggest that larger programs could be handled. This is not to say, of course, that production-size programs could be verified or synthesized. To do this, much work on search strategies, proof management and domain knowledge would be required. This is an obvious and important area of further work.

In the examples we have given, we have confined ourselves to a programming language equivalent to pure Prolog. The formalism permits extensions to this. It would be possible, for example, to derive programs with negation without changing anything in the basic framework. To draw upon a larger class of partial inductive definitions for programs, we would need an efficient proof procedure for such programs. Several efficient implementations of Prolog with negation exist (e.g. [15]), and at SICS work is being done on efficient proof procedures for general partial inductive definitions [2] [7].

The major obstacle to the use of larger program classes is that the conservativity and monotonicity properties must be determined. These properties are impossible to give criteria for in general, and it remains to be investigated for what interesting subsets of partial inductive definitions such criteria can be given.

The generality offered by the choice of specification language has not been explored. We intend to try some different logics and possibly some completely different languages, to determine how general the approach really is.

Both inductive definitions and derivations are infinite objects in general, and finite representations of them are needed. In this paper we have used informal and simple notations for these purposes. We are currently working on a general and precise representation to be used to write down and develop derivations.

10. REFERENCES

[1] Aczel, Peter, *An Introduction to Inductive Definitions*, in: Handbook of Mathematical Logic (Barwise, J., ed.), North-Holland, Amsterdam 1977.

[2] Aronsson, Martin, *An Abstract Machine for Generalized Horn Clauses*, SICS Research Report to be published, Swedish Institute of Computer Science, Stockholm.

[3] Clark, Keith and Tärnlund, Sten-Åke, *A First Order Theory of Data and Programs*, Proceedings of IFIP -77 pp. 939-944, North-Holland, Amsterdam.

[4] Eriksson, Lars-Henrik, *Derivation of a Unification Algorithm in the Logic Programming Calculus*, The Journal of Logic Programming 1 (1) pp. 3-18, 1984.

[5] Hallnäs, Lars, *A Note on the Logic of a Logic Program*, in: Proceedings of the Workshop on Programming Logic, report PMG-R37, University of Göteborg and Chalmers University of Technology, 1987.

[6] Hallnäs, Lars, *Partial Inductive Definitions*, SICS Research Report R86005B, Swedish Institute of Computer Science, Stockholm 1987.

[7] Hallnäs, Lars and Schroeder-Heister, Peter, *A Proof-Theoretic Approach to Logic Programming I. Generalized Horn Clauses*, SICS Research Report R88005, Swedish Institute of Computer Science, Stockholm 1988.

[8] Hansson, Åke, *A Formal Development of Programs*, Ph.D. thesis, Department of Information Processing and Computer Science, University of Stockholm, Stockholm 1980.

[9] Hansson, Åke and Tärnlund, Sten-Åke, *Program Transformation by Data Structure Mapping*, in: Logic Programming (Clark, K. and Tärnlund S.-Å., eds.) pp. 117-122, Academic Press, London 1982.

[10] Hogger, Christopher John, *Derivation of Logic Programs*, Journal of the ACM 28 (2) pp. 372-392, 1981

[11] Hogger, Christopher John, *Introduction to Logic Programming*, Academic Press, London 1984.