Program Semantics and Classical Logic

Reinhard Muskens

Abstract

In the tradition of Denotational Semantics one usually lets program constructs take their denotations in reflexive domains, i.e. in domains where self-application is possible. For the bulk of programming constructs, however, working with reflexive domains is an unnecessary complication. In this paper we shall use the domains of ordinary classical type logic to provide the semantics of a simple programming language containing choice and recursion. We prove that the rule of *Scott Induction* holds in this new setting, prove soundness of a Hoare calculus relative to our semantics, give a direct calculus $\mathcal{C}$ on programs, and prove that the denotation of any program P in our semantics is equal to the union of the denotations of all those programs L such that P follows from L in our calculus and L does not contain recursion or choice.

1 Introduction

In this paper we translate a simple but non-trivial imperative programming language into an axiomatic extension of classical second order logic. Since classical logic comes with a modeltheoretic interpretation, we get an interpretation for our programming language too, as we may identify the denotation of any programming construct in a given model with the denotation of its translation in that model. The resulting semantics thus assigns a mathematical object to each programming construct in each model.

This last aspect makes the paper fall within the tradition of *Denotational Semantics* [21, 20, 3, 19, 13, 22]. But we shall deviate considerably from the usual denotational approach by not making any use of the *Scott domains* which are ubiquitous in that tradition. A characteristic property of such domains is that they can be *reflexive,* i.e. a domain D can be isomorphic to, and can in fact be identified with, the function space $[D \to D]$ of all continuous functions from D to D. This guarantees that any object $d \in D$ is also a function $d : D \to D$ and hence that it is meaningful to talk about $d(d)$. Scott domains thus support the interpretation of self-application and in fact are essential for the interpretation of functional languages which are based on the untyped lambda calculus and in which self-application is possible. But, although there are some imperative languages which allow self-application, it seems that the bulk of constructs normally encountered in imperative languages, including iteration and recursion, can be treated without the use of reflexive domains. As we shall see in this paper, imperative programs containing iteration and recursion can be interpreted in the models of classical type logic via a translation into the second-order fragment of that logic.

Classical type logic itself then can function as a kind of universal specification language in the design of programming languages: In order to specify the intended semantics of some construct, the language designer may simply write down its intended logical translation. On a more theoretical level, we consider it an advantage to be able to describe the semantics of programming constructs with the help of the language and the models which also underly mainstream mathematical logic. True, there are important differences between the semantics of mathematics and the semantics of programming, in the sense that the model theory of, say, the natural numbers is 'static', while the model theory of a program must needs be 'dynamic'. But these differences, important as they are, should not blind us to the fact that the two forms of semantics are essentially one. We shall model the difference between 'static' and 'dynamic' semantics as a difference in *type* here. While static theories are sets of sentences, i.e. closed terms of type t, the logical translation of a program must have a more complex type. If we choose to treat programs as relations between input and output states, as we shall do here (other choices are very well possible), the type of programs becomes $s \times s \to t$. It is this type difference which constitutes the difference between static and dynamic semantics, but the type difference is already present in standard logic.

Treating the semantics of a programming language with the help of *classical* logic should not be construed as being in opposition to the effort of *Dynamic Logic* ([18, 7]), the *modal* logic of programs, however. In Dynamic Logic one typically studies logics containing only a very few programming constructs. The goals here are of a purely logical nature and the primary interest is in stating and proving metatheorems on completeness, decidability etc. Such aims are best served with logics with a limited expressivity, as increase in expressivity generally leads to loss of metalogical properties. On the other hand, the goal of providing a general logical framework for program semantics, as we find it in denotational semantics, does not square well with such limited expressivity. In this area and for this purpose we must have rich and typed logics. Our decision to use classically typed models instead of the domains one usually finds in denotational semantics brings us closer to the dynamic enterprise, but we still need a logic which is more expressive than the ones which are characteristically studied in that tradition. The goals and aims of Dynamic Semantics and Denotational Semantics are of a complementary rather than of an opposing nature and in fact the present logic can be viewed as a typed extension of Quantificational Dynamic Logic (QDL): see [14, 17] for an embedding of QDL into (a variant of) the present system which is related to the well-known embedding of PDL into $L_{\omega_1\omega}$.

Giving the semantics of a language by means of a compositional translation into some typed logic is a procedure which is known in linguistics under the name of *Montague Semantics* [11, 12]. Linguists try to explain the semantics of, say, English by translating fragments of that language into suitable higher-order logics. We do the same here for a simple programming language, thereby threading into the footsteps of Janssen [10], who gave a Montague Semantics for programming by translating fragments of an Algol-like language into a special *Dynamic Intensional Logic*. Janssen did not give a treatment of recursion and we hope to improve upon his work by repairing this omission. Janssen's logic has also played a role in linguistics, as Groenendijk and Stokhof have applied it to this area with some success in [6]. Variants of the axiomatic extension of classical logic that will be used in this paper have been used by the present author for giving the semantics of fragments of English in [14, 15, 16] ([14] also considers a treatment of the `while` languages). A related system, which uses a non-classical set-up, is Van Eijck's interesting paper [26]. Van Eijck prefers to replace the non-logical axioms of [14] by certain extralogical requirements on his models, but proves that some important properties of standard logic are not lost.

The set-up of the rest of this paper is as follows. In section 2 we shall define a simple programming language with choice and recursion, give some abbreviations and introduce some necessary syntactic conventions. In section 3 then, we present a calculus $\mathcal{C}$ in which certain rewriting operations on programs are possible. These rewritings are in close correspondence to the intended behaviour of our programs and in fact $P_1 \vdash_{\mathcal{C}} P_2$ will imply that an execution of P_1 is also a possible way of executing P_2. Section 4 gives a short overview of classical type logic (of which we shall use only the second-order part) and section 5 introduces the necessary axiomatic extension. One restriction on our class of models will be that we shall want a certain domain D_e to behave like the natural numbers, to which end we shall adopt the (second-order) Peano axioms. Another constraint will be that we want a domain D_s to act like a set of *program states*. This will be achieved by adopting a first-order axiom. The actual translation of the programming language then follows in section 6. Section 7 imports the concepts of *continuity, anticontinuity* and *monotonicity* from the usual Scott approach, develops some theory, and proves that Scott's Induction Rule holds within the present setting. Section 8 introduces the usual Hoare calculus for our programming language and proves it to be sound relative to the semantics of section 6. The main theorem of the paper then follows in section 9: We define a *linear* program as a program which is just a sequence of atomic statements and, for any program P, consider all linear programs L such that $L \vdash_{\mathcal{C}} P$. It turns out that the denotation of P, given our semantics, is exactly the union of the denotations of all such L.

2 A Toy Programming Language

The programming language which we shall study in this paper is defined by the Backus-Naur form in definition 1, which is almost self-explanatory. The set Num consists of the numerals 0, $S(0)$, $S(S(0))$,..., which we shall write in decimal notation. N consists of arithmetical terms built out of numerals and first-order variables. In B we find Boolean combinations of equalities and inequalities built up from these terms, and F is essentially the language of first order arithmetic. The set A consists of *assignment statements* $x := N$, *tests* $B?$, and *program variables,* or *procedure calls* X. The category P extends this set by allowing *sequencing, choice* and, the most important programming construct in this paper, *recursion.* We allow a set of procedure declarations $\langle X_1 \Leftarrow P_1, \ldots, X_n \Leftarrow P_n \rangle$, abbreviated $\langle X_i \Leftarrow P_i \rangle_{i=1}^n$, or even $\langle X_i \Leftarrow P_i \rangle_i$, to bind the program variables $X_1, \ldots, X_n$ in a given program P. The variables $X_1, \ldots, X_n$ may occur free (for the definition of free occurrence see below) in $P_1, \ldots, P_n$, so that procedure declarations may call one another.

Definition 1 (Language)

$$\begin{array}{rcl}
Num & ::= & 0 \quad | \quad S(Num) \\
N & ::= & x \quad | \quad Num \quad | \quad N_1 + N_2 \quad | \quad N_1 \times N_2 \\
B & ::= & \texttt{false} \quad | \quad N_1 = N_2 \quad | \quad N_1 \leq N_2 \quad | \quad B_1 \to B_2 \\
F & ::= & B \quad | \quad F_1 \to F_2 \quad | \quad \forall x F \\
A & ::= & x := N \quad | \quad B? \quad | \quad X \\
P & ::= & A \quad | \quad P_1 ; P_2 \quad | \quad P_1 \cup P_2 \quad | \quad \langle X_i \Leftarrow P_i \rangle_{i=1}^{n}(P) \\
C & ::= & \{F_1\} P \{F_2\} \quad | \quad P_1 \subseteq P_2
\end{array}$$

Note that the construction $\langle X_i \Leftarrow P_i \rangle_{i=1}^{n}(P)$, unlike the similar construct studied in [3], may be nested to an arbitrary depth. A real programming language such as PASCAL would write $\langle X_i \Leftarrow P_i \rangle_{i=1}^{n}(P)$ as follows.

`procedure` X_1;
P_1;
$\vdots$
`procedure` X_n;
P_n;
P

This exhausts our stock of programs. The category C finally, consists of *correctness statements,* which can be divided into *asserted programs* $\{F_1\}P\{F_2\}$ stating that program P, whenever it is started in a state where F_1 holds, will after any successful execution be in a state where F_2 holds. The statement $P_1 \subseteq P_2$ expresses that if we can reach a state j starting from a state i by running P_1, we can also get from i to j by running P_2.

The following abbreviations are useful.

Definition 2 (Abbreviations)

$$\begin{array}{rll}
\texttt{true} & \textit{is short for} & \texttt{false} \to \texttt{false} \\
\neg F & \textit{is short for} & F \to \texttt{false} \\
F_1 \vee F_2 & \textit{is short for} & \neg F_1 \to F_2 \\
F_1 \wedge F_2 & \textit{is short for} & \neg(\neg F_1 \vee \neg F_2) \\
\exists x F & \textit{is short for} & \neg \forall x \neg F \\
N_1 < N_2 & \textit{is short for} & N_1 \leq N_2 \wedge \neg N_1 = N_2
\end{array}$$

$$
\begin{array}{rll}
\texttt{skip} & \textit{is short for} & \texttt{true}? \\
\texttt{fail} & \textit{is short for} & \texttt{false}? \\
\texttt{if } B \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi} & \textit{is short for} & (B?; P_1) \cup (\neg B?; P_2) \\
\mu X(P) & \textit{is short for} & \langle X \Leftarrow P \rangle (X) \\
\texttt{while } B \texttt{ do } P \texttt{ od} & \textit{is short for} & \mu X((B?; P; X) \cup \neg B?)
\end{array}
$$

Remark on notation. The sequence operator ";" is associative, both on its intuitive interpretation ($P_1; P_2$ meaning "do P_1 and then P_2") and given the semantics (relational composition) it gets in this paper and elsewhere. We shall also make no *syntactic* distinction between $(P_1; P_2); P_3$ and $P_1; (P_2; P_3)$ and write both as $P_1; P_2; P_3$. Similarly, we syntactically identify P with `skip`; P and with P;`skip`. This is allowed because `skip` intuitively means "do nothing" and will be interpreted formally as the *diagonal* relation $\{\langle i, i \rangle \mid$ i is a state$\}$. The identification will allow us to say that the programs $P_1; P$ and $P; P_2$, as well as the program P, have the form $P_1; P; P_2$: we can take P_1 or P_2 or both to be the identity element `skip`.

An occurrence of a program variable $Y \in \{X_1, \ldots, X_n\}$ is *free* in a program P if it does not occur within a subprogram of the form $\langle X_i \Leftarrow P_i \rangle_{i=1}^n (P')$ in P. A program is *closed* if it contains no free program variables.

Our procedure declarations can bind a series of program variables simultaneously, and since procedure declarations may make mutual calls it will not be possible to reduce this form of binding to an *iteration* of bindings, as it can usually be done in logic. Since we shall also have occasion to work with simultaneous substitutions, things threaten to get a bit complicated. The following definition may look a bit hairy, but in fact gives a straightforward generalisation of the usual notion of substitution.

Definition 3 (Substitution) *The* simultaneous substitution $[P_i/X_i]_{i=1}^n P$ of P_1 for X_1 and ... and P_n for X_n in P *(abbreviated as $[P_i/X_i]_i P$) is defined as follows.*

$$
\begin{array}{rcl}
[P_i/X_i]_i X_k & = & P_k, \ \textit{if } 1 \leq k \leq n \\
[P_i/X_i]_i A & = & A, \ \textit{if } A \notin \{X_1, \ldots, X_n\} \\
[P_i/X_i]_i (Q_1; Q_2) & = & [P_i/X_i]_i \, Q_1; [P_i/X_i]_i \, Q_2 \\
[P_i/X_i]_i (Q_1 \cup Q_2) & = & [P_i/X_i]_i \, Q_1 \cup [P_i/X_i]_i \, Q_2
\end{array}
$$

$$[P_i/X_i]_i\langle Y_k \Leftarrow Q_k\rangle_{k=1}^m(Q) \quad = \quad \langle Z_k \Leftarrow Q'_k\rangle_{k=1}^m(Q'), \textit{ where } Z_1,\ldots,Z_m \textit{ are fresh, } Q'_k \textit{ abbreviates } [P_i/X_i]_i\,[Z_\ell/Y_\ell]_{\ell=1}^m\,Q_k \textit{ and } Q' \textit{ abbreviates } [P_i/X_i]_i\,[Z_\ell/Y_\ell]_{\ell=1}^m\,Q$$

An alternative notation for $[P_i/X_i]_{i=1}^n P$ is $[P_1/X_1,\ldots,P_n/X_n]P$. The following lemma is standard and has a straightforward proof.

Lemma 1 *If $\{X_1,\ldots,X_n\}\cap\{Y_1,\ldots,Y_m\}=\emptyset$ and $Y_1,\ldots,Y_m$ are not free in $P_1,\ldots,P_n$, then*

$$[P_i/X_i]_{i=1}^n\,[Q_k/Y_k]_{k=1}^m\,P = [[P_i/X_i]_{i=1}^n\,Q_k/Y_k]_{k=1}^m\,[P_i/X_i]_{i=1}^n\,P$$

3 A Computational Calculus

Let us call a program which consists only of a sequence of assignments $x := N$, tests $B?$ and program variables X *linear*, so that the class L of linear programs is given by

$L \quad ::= \quad A \quad | \quad L_1; L_2$

We present a simple calculus $\mathcal{C}$ characterising a derivability relation $\vdash_{\mathcal{C}}$ on the set of programs. The idea is that if $L \vdash_{\mathcal{C}} P$ holds for some closed L, then an execution of the deterministic L will count as one possible execution of the possibly indeterministic P. Conversely, any execution of P will be an execution of some L such that $L \vdash_{\mathcal{C}} P$. The rules of the calculus are the following.

$$\frac{P_i}{P_1 \cup P_2} \quad i \in \{1,2\} \qquad\qquad (\cup \text{ rule})$$

$$\frac{[\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_1)/X_1,\ldots,\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_n)/X_n]P}{\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P)} \qquad (\rho \text{ rule})$$

The idea behind these rules is that bottom-up they can be read as rules for executing a program. Executing $P_1 \cup P_2$ consists of executing P_1 or executing P_2 and an execution of $\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P)$ consists of replacing all procedure calls X_k in P by their bodies P_k, but so that further procedure calls in P_k are still bound by the delarations in $\langle X_i \Leftarrow P_i\rangle_{i=1}^n$. This means that we have to substitute the X_k in P simultaneously by $\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)$. Viewed in this procedural way, the rule is a generalisation of the procedural rule for the μ operator studied in [21, 3].

The following definition says what it is for a program to follow from another program in one step. Note that we make use of the notational convention given in the previous section here: Q_1 and Q_2 may be taken to be empty.

Definition 4 (Immediate Consequence) *A program $Q_1; Q; Q_2$ is an* immediate consequence *of a program $Q_1; Q'; Q_2$ iff Q follows from Q' by the $\cup$ rule or the ρ rule. $Q_1; Q; Q_2$ is a* leftmost *immediate consequence of $Q_1; Q'; Q_2$ if, moreover, Q_1 is linear.*

From the notion of immediate consequence we can define the notion of computational entailment we are after.

Definition 5 (Computational Entailment) *A* (leftmost) derivation of P_n from P_1 *is a sequence of programs $P_1, \ldots, P_n$ such that each P_{i+1} is a (leftmost) immediate consequence of P_i. We write $P \vdash_{\mathcal{C}} Q$ ($P \vdash_{\mathcal{LC}} Q$) iff there is a (leftmost) derivation of Q from P.*

Example 1 The following is a leftmost derivation of

$$\langle Z \Leftarrow (x < 3?; x := x + 1; Z) \cup \neg x < 3? \rangle (Z),$$

which we shall abbreviate as $\mathcal{D}(Z)$, from the linear program $x < 3?; x := x + 1; x < 3?; x := x + 1; \neg x < 3?$.

1. $x < 3?; x := x + 1; x < 3?; x := x + 1; \neg x < 3?$ (premise)
2. $x < 3?; x := x + 1; x < 3?; x := x + 1;$ $((x < 3?; x := x + 1; \mathcal{D}(Z)) \cup \neg x < 3?)$ ($\cup$ rule)
3. $x < 3?; x := x + 1; x < 3?; x := x + 1; \mathcal{D}(Z)$ (ρ rule)
4. $x < 3?; x := x + 1; ((x < 3?; x := x + 1; \mathcal{D}(Z)) \cup \neg x < 3?)$ ($\cup$ rule)
5. $x < 3?; x := x + 1; \mathcal{D}(Z)$ (ρ rule)
6. $(x < 3?; x := x + 1; \mathcal{D}(Z)) \cup \neg x < 3?$ ($\cup$ rule)
7. $\mathcal{D}(Z)$ (ρ rule)

Note that $\mathcal{D}(Z)$ is the expanded form of `while` $x < 3$ `do` $x := x + 1$ `od`. Clearly, the set of linear programs L such that $L \vdash_{\mathcal{LC}} \mathcal{D}(Z)$ can informally be described as $(x < 3?; x := x + 1)^*; \neg x < 3?$, where * is the Kleene star.

We shall be interested in the set of linear programs L which derive a given program P. The following proposition shows that it is immaterial whether we take the notion $\vdash_{\mathcal{C}}$ or the notion $\vdash_{\mathcal{LC}}$ as the underlying entailment relation. Leftmost derivations are closer to actual executions, but the order in which we execute statements does not matter for our purposes.

Proposition 2 *If $L \vdash_{\mathcal{C}} P$ then $L \vdash_{\mathcal{LC}} P$*

Proof. Let $L \vdash_{\mathcal{C}} P$. We use induction on the length n of the derivation $L = P_1, \ldots, P_n = P$ of P from L to show that a leftmost derivation of P from L of length n can be constructed. If $n = 1$ then $P = L$ and the derivation is leftmost, so assume that $n > 1$. Then P_n is of the form $L_0; Q; Q_n$ with Q of one of the forms $P' \cup P''$ or $\langle X_i \Leftarrow P_i \rangle_{i=1}^n(P')$. Consider the greatest i such that P_i does not have the form $L_0; Q; Q_i$. Then P_k is of the form $L_0; Q; Q_k$ for $i < k \leq n$, while P_i has the form $L_0; Q'; Q_{i+1}$, Q follows from Q' by the $\cup$ rule or the ρ rule, and P_{i+1} is a leftmost immediate consequence of P_i. For $i < k \leq n$, define P'_k to be $L_0; Q'; Q_k$. Then $P_i = P'_{i+1}$ and it is easily seen that $P_1, \ldots, P_{i-1}, P'_{i+1}, \ldots, P'_n$ is a derivation of P'_n from L of length $n - 1$. Using induction we find a leftmost derivation of P'_n from L of the same length. Since P_n is a leftmost immediate consequence of P'_n $(= L_0; Q'; Q_n)$ this also gives us a leftmost derivation of P from L of length n. $\square$

4 Classical Type Logic

Mainly in order to fix notation, we give a short exposition of classical type logic, Church's formulation of Russell's *Simple Theory of Types*. For more extensive treatments see [2, 8, 9, 4, 25, 1].

In classical type logic (higher-order logic) every expression comes with a *type*. Types are either basic or complex. The type t of truth values should be among the basic types, but there may be other basic types as well. In this paper we fix the set of basic types as $\{t, e, s\}$, where e is the type of *natural numbers* and s the type of *states*. The logic we are describing here thus is essentially the two-sorted type logic TY_2 of [4] (but we shall add product types).

We build up complex types in the following way.

Definition 6 (Types) *The set of* types *is the smallest set such that:*

1. *all basic types are types;*
2. *if α and β are types then $\alpha \to \beta$ and $\alpha \times \beta$ are types.*

Each type may correspond to a set of objects of that type. In this way hierarchies of type domains, called *(standard) frames* are obtained.

Definition 7 (Standard Frames) *A* (standard) frame *is a set of non-empty sets $\{D_\alpha \mid \alpha \text{ is a type}\}$ such that, for all α, β*

$$\begin{array}{rcl} D_t & = & \{0,1\}; \\ D_{\alpha\to\beta} & = & D_\alpha \to D_\beta; \\ D_{\alpha\times\beta} & = & D_\alpha \times D_\beta. \end{array}$$

Frames will be the backbones of our models. We shall use them in order to interpret the language which we are about to define. For each type α, let us assume the existence of a denumerably infinite set of variables VAR_α and some set of constants CON_α. From these basic terms complex ones can be built.

Definition 8 (Terms) *Define, for each α, the set $\mathcal{T}_\alpha$ of* terms *of type α by the following induction.*

1. $CON_\alpha \subseteq \mathcal{T}_\alpha$, $VAR_\alpha \subseteq \mathcal{T}_\alpha$;
2. $\mathcal{A} \in \mathcal{T}_{\alpha\to\beta}$, $\mathcal{B} \in \mathcal{T}_\alpha \Rightarrow \mathcal{A}(\mathcal{B}) \in \mathcal{T}_\beta$;
3. $\mathcal{A} \in \mathcal{T}_\beta$, $\xi \in VAR_\alpha \Rightarrow \lambda\xi.\mathcal{A} \in \mathcal{T}_{\alpha\to\beta}$;
4. $\mathcal{A} \in \mathcal{T}_\alpha$, $\mathcal{B} \in \mathcal{T}_\beta \Rightarrow \langle\mathcal{A},\mathcal{B}\rangle \in \mathcal{T}_{\alpha\times\beta}$;
5. $\mathcal{A} \in \mathcal{T}_{\alpha\times\beta} \Rightarrow (\mathcal{A})_0 \in \mathcal{T}_\alpha$ *and* $(\mathcal{A})_1 \in \mathcal{T}_\beta$;
6. $\mathcal{A} \in \mathcal{T}_\alpha$, $\mathcal{B} \in \mathcal{T}_\alpha \Rightarrow (\mathcal{A} = \mathcal{B}) \in \mathcal{T}_t$.

If $\mathcal{A} \in \mathcal{T}_\alpha$ we may (but need not) indicate this by writing $\mathcal{A}_\alpha$. Terms of type t are called *formulae*. In $(\mathcal{A})_0$ and $(\mathcal{A})_1$ the parentheses are often omitted if this is not likely to lead to confusion. Note that the first five clauses in the above definition give the syntax of a simply typed λ-calculus with pairing and projection, but that the sixth clause explicitly requires the logical operator $=$ to be in the language. Having $=$ is enough to get the usual other logical operators, as some crucial ones can be defined as follows (see [9]) and the others can be obtained as in Definition 2.

Definition 9 (Abbreviations)

$$\begin{array}{rll}
\forall\xi_\alpha.\,\varphi & \textit{is short for} & (\lambda\xi_\alpha.\,\varphi) = (\lambda\xi_\alpha.\,\xi = \xi) \\
\texttt{false} & \textit{is short for} & \forall\xi_t.\,\xi_t \\
\varphi \to \psi & \textit{is short for} & (\langle \texttt{true}, \texttt{false}\rangle = \langle\varphi,\psi\rangle) = \texttt{false}
\end{array}$$

In order to interpret the language of Definition 8, we need to have interpretations of its constants and variables. An interpretation function $\mathcal{I}$ for a frame $\{D_\alpha\}_\alpha$ is a function with the set of all constants as its domain such that $\mathcal{I}(c_\alpha) \in D_\alpha$ for all c_α. Similarly, an assignment a for $\{D_\alpha\}_\alpha$ is a function taking variables as its arguments such that $a(\xi_\alpha) \in D_\alpha$ for all ξ_α. We write $a[d_1/\xi_1, \ldots, d_n/\xi_n]$ for the assignment a' such that $a'(\xi_i) = d_i$ if $1 \leq i \leq n$ and $a'(\xi) = a(\xi)$ if $\xi \notin \{\xi_1, \ldots, \xi_n\}$. A *(standard) model* is a pair $\langle\mathcal{F},\mathcal{I}\rangle$ of a frame $\mathcal{F}$ and an interpretation function $\mathcal{I}$ for $\mathcal{F}$. The following definition provides the logic with a semantics. (We use `fst` and `snd` as functions that will pick out the first and second element of an ordered pair respectively.)

Definition 10 (Tarski Definition) *The* value $\mathcal{V}^{\mathcal{M}}(\mathcal{A}, a)$ *of a term $\mathcal{A}$ on a standard model $\mathcal{M} = \langle\mathcal{F},\mathcal{I}\rangle$ under an assignment a for $\mathcal{F}$ is defined as follows (we suppress superscripts $\mathcal{M}$ to improve readability).*

1. $\mathcal{V}(c, a) = \mathcal{I}(c)$, *if c is a constant;*
   $\mathcal{V}(\xi, a) = a(\xi)$, *if ξ is a variable;*
2. $\mathcal{V}(\mathcal{A}(\mathcal{B}), a) = \mathcal{V}(\mathcal{A}, a)(\mathcal{V}(\mathcal{B}, a))$, *i.e.* $\mathcal{V}(\mathcal{A}, a)$ *applied to* $\mathcal{V}(\mathcal{B}, a)$;
3. $\mathcal{V}(\lambda\xi_\alpha.\,\mathcal{A}_\beta, a) =$ *the $G \in D_{\alpha\to\beta}$ such that, for each $d \in D_\alpha$,*
   $G(d) = \mathcal{V}(\mathcal{A}, a[d/\xi])$;
4. $\mathcal{V}(\langle\mathcal{A},\mathcal{B}\rangle, a) = \langle\mathcal{V}(\mathcal{A}, a), \mathcal{V}(\mathcal{B}, a)\rangle$;
5. $\mathcal{V}((\mathcal{A})_0, a) = \texttt{fst}(\mathcal{V}(\mathcal{A}, a))$; $\quad\mathcal{V}((\mathcal{A})_1, a) = \texttt{snd}(\mathcal{V}(\mathcal{A}, a))$;
6. $\mathcal{V}(\mathcal{A} = \mathcal{B}, a) = 1$ iff $\mathcal{V}(\mathcal{A}, a) = \mathcal{V}(\mathcal{B}, a)$.

Note that the logical operators that were obtained in Definition 9 get their standard meaning.

We can now define the notion of entailment.

Definition 11 *Let* $\Gamma \cup \{\varphi\}$ *be a set of sentences.* Γ (standardly) entails φ, $\Gamma \models_s \varphi$, *iff, for each model* $\mathcal{M}$, *and each assignment* a *for* $\mathcal{M}$, $\mathcal{V}^{\mathcal{M}}(\varphi, a) = 1$, *if* $\mathcal{V}^{\mathcal{M}}(\gamma, a) = 1$ *for all* $\gamma \in \Gamma$.

It will be useful to have a typographical convention which helps us distinguish between variables of various types. The following table gives an overview of the symbols we shall typically use for variables of a fixed type. For variables whose type is not fixed, we shall continue to use ξ and ζ.

Type	Variables
e	x, y, z, v
$e \to t$	$\mathcal{X}$
s	i, j
$s \times s$	r
$s \times s \to t$	X, Y

5 Nonlogical Axioms

There are two kinds of constraints we wish to impose on our models. First, we want our domain D_s to behave as a set of states. Second, we want D_e to behave as the natural numbers.

Starting with the second requirement, we impose the (second order) Peano Axioms, which for the sake of completeness we shall list here. We refer to the conjunction of these axioms as PA^2.

$\neg\exists x.\, S(x) = 0$
$\forall x\, \forall y.\, S(x) = S(y) \to x = y$
$\forall x.\, x + 0 = x$
$\forall x\, \forall y.\, x + S(y) = S(x + y)$
$\forall x.\, x \times 0 = 0$
$\forall x\, \forall y.\, x \times S(y) = (x \times y) + x$
$\forall \mathcal{X}\, ((\mathcal{X}(0) \land \forall x\, (\mathcal{X}(x) \to \mathcal{X}(S(x)))) \to \forall x\, \mathcal{X}(x))$

Here the constants $+$ and $\times$ are of type $e \times e \to e$, but we use infix notation and write $x + y$ and $x \times y$ for $+(\langle x, y\rangle)$ and $\times(\langle x, y\rangle)$ respectively. As is usual, we abbreviate $\exists z.\, x + z = y$ as $x \leq y$ and we write $S^0(0)$ for 0 and $S^{n+1}(0)$ for $S(S^n(0))$, so that $S^n(0)$ is be the object-level representation of the number n.

We identify the variables x introduced in section 2 with the variables of type e and variables X with the variables of type $s \times s \to t$. Note that the first identification turns the set F defined in section 2 into a subset of the set of first-order formulae of our logic.

The natural numbers shall in fact play a double role in our translation of the programming language defined in section 2. Their first role is obvious: they must interpret the elements of N. Their second role will be to act as *addresses of registers*. In order to let them behave in this way, we let V be some fixed non-logical constant of type $e \to (s \to e)$. The intended interpretation of, say, $V(S^{17}(0))(i)$ is the number which is stored in the register with address 17 in state i. The formula $V(S^{17}(0))(i) = S^{23}(0)$ thus expresses that the value of register 17 in state i is 23. As a mnemonic to help distinguish between these two roles of the natural numbers, we shall use the variable v whenever we think of them as the addresses of registers but use x, y, z in all other cases. Formally, however, there is no difference here.

Since we want objects of types s to really behave as states, and since we want it to be the case that registers can be updated independently from each other, we shall impose an axiom. If t is a term of type e and i and j are variables of type s we may use $i[t]j$ to abbreviate $\forall v(v \neq t \to V(v)(i) = V(v)(j))$. This expresses that states i and j differ at most in the register with address t. Our axiom requires that there are enough states; so that given any state, any register, and any number, we can update the register and set its value to the given number without effecting the values of other registers.[1]

UA $\quad \forall i\, \forall v\, \forall x\, \exists j.\, i[v]j \wedge V(v)(j) = x$

[1]If we interpret registers as (logical) variables, states as (logical) assignments, and $V(v)(i)$ as the application of assignment i to variable v, the axiom formalises the usual situation in logic that we have enough assigments to let variables take their values independently from one another. In [23, 24] Van Benthem studies various weakenings of this requirement, obtaining variants of predicate logic which are decidable and in which one can have *dependent variables*. Weakening UA would not result in decidability of course, but would also result in introducing similar dependencies among variables. This could be useful for particular purposes. See also [17].

We shall refer to UA as to the *Update Axiom*. The axiom is related to the 'Update Postulate' of [10] and to the 'Having Enough States' axiom of [5]. We write $\Gamma \models_{AX} \varphi$ if Γ, UA, $\mathrm{PA}^2 \models_s \varphi$.

This ends the list of axioms we wish to impose. To see that they are consistent, it suffices to construct a model in which D_e consists of the set $\mathcal{N}$ of natural numbers, where D_s is the set of all functions $f : \mathcal{N} \to \mathcal{N}$, and where $\mathcal{I}(V)$ is the function such that $\mathcal{I}(V)(n)(f) = f(n)$ for all $n \in D_e$ and all $f \in D_s$. We can obtain a model with denumerable basic domains if we let D_s be the set of all functions $f : \mathcal{N} \to \mathcal{N}$ such that $f(n) \neq 0$ for finitely many n and define D_e and $\mathcal{I}(V)$ as before.

6 The Translation

We are now in the position to define the promised translation of our programming language into the second-order part of classical type logic. We shall give it in three installments, translating terms and formulas first and programs and correctness formulas afterwards. We assume that VAR_e is enumerated by the natural numbers. In the following definition arithmetical terms N translate as terms of type $s \to e$ and formulas F translate as terms of type $s \to t$.

Definition 12 (Translation of Terms and Formulas)

$$
\begin{aligned}
0^\dagger &= \lambda i.0 && (1)\\
(S(Num))^\dagger &= \lambda i.S(Num^\dagger(i)) && (2)\\
x_k^\dagger &= \lambda i.V(S^k(0))(i) && (3)\\
(N_1 + N_2)^\dagger &= \lambda i.N_1^\dagger(i) + N_2^\dagger(i) && (4)\\
(N_1 \times N_2)^\dagger &= \lambda i.N_1^\dagger(i) \times N_2^\dagger(i) && (5)\\
\mathtt{false}^\dagger &= \lambda i.\mathtt{false} && (6)\\
(N_1 = N_2)^\dagger &= \lambda i.N_1^\dagger(i) = N_2^\dagger(i) && (7)\\
(N_1 \leq N_2)^\dagger &= \lambda i.N_1^\dagger(i) \leq N_2^\dagger(i) && (8)\\
(F_1 \to F_2)^\dagger &= \lambda i.F_1^\dagger(i) \to F_2^\dagger(i) && (9)\\
(\forall x_k F)^\dagger &= \lambda i.\, \forall j\ (i[S^k(0)]j \to F^\dagger(j)) && (10)
\end{aligned}
$$

Most clauses in this definition are self-explanatory; but note that in (3) variables x are translated as the values of registers and that quantification over individual variables is translated in (10) by means of quantification over states. The following two Lemmas are there to ensure that this is correct.

Lemma 3 *Let φ be a formula in which j does not occur free, then*

$$\models_{AX} \forall v \forall i\, ((\forall j\, (i[v]j \to [V(v)(j)/x]\varphi) \leftrightarrow \forall x\, \varphi)$$

Proof. Immediate from AX1. □

Lemma 4 *For any type s variable i and term N, let N^i be obtained by replacing, for all k, each free x_k in N with $V(S^k(0))(i)$. Similarly, let F^i be the result of replacing each free x_k in F with $V(S^k(0))(i)$ for all k. Then*

1. $\models_{AX} N^{\dagger}(i) = N^i$
2. $\models_{AX} F^{\dagger}(i) \leftrightarrow F^i$

Proof. The truth of the first statement is obvious and the second statement is proved by an induction on the complexity of formulas in which only the $\forall x_k F$ case is interesting. For the latter, consider that, by Lemma 3, $(\forall x_k F)^i$ is equivalent with $(\forall j\, (i[S^k(0)]j \to [V(S^k(0))(j)/x_k]F))^i$ and use the definition of $i[S^k(0)]j$ to see that this formula is equivalent with $\forall j\, (i[S^k(0)]j \to F^j)$. Induction tells us that this formula in its turn is equivalent with $\forall j\, (i[S^k(0)]j \to F^{\dagger}(j))$, which needed to be proved. □

The previous Lemma in fact tells us that the translation given thus far preserves entailment between formulas F if we take our axioms into account. The following definition translates programs into type logic. The type of target terms is $s \times s \to t$ here: binary relations between states.

Definition 13 (Translation of Programs)

$$\begin{aligned}
(x_k := N)^{\dagger} &= \lambda r.\, r_0[S^k(0)]r_1 \wedge V(S^k(0))(r_1) = N^{\dagger}(r_0) && (11)\\
(B?)^{\dagger} &= \lambda r.\, B^{\dagger}(r_0) \wedge r_0 = r_1 && (12)\\
X^{\dagger} &= X && (13)\\
(P_1; P_2)^{\dagger} &= \lambda r.\, \exists j.\, P_1^{\dagger}(\langle r_0, j\rangle) \wedge P_2^{\dagger}(\langle j, r_1\rangle) && (14)\\
(P_1 \cup P_2)^{\dagger} &= \lambda r. P_1^{\dagger}(r) \vee P_2^{\dagger}(r) && (15)\\
(\langle X_i \Leftarrow P_i\rangle_{i=1}^{n}(P))^{\dagger} &= \lambda r.\, \forall X_1 \ldots X_n (\bigwedge_{i=1}^{n} X_i = P_i^{\dagger} \to P^{\dagger}(r)) && (16)
\end{aligned}$$

We see here that an assignment statement $x := N$ is interpreted as the set of pairs of input and output states such that input and output state differ at most in the affected register and the value of the register in the output state is the original value of N. The translation is clearly dependent upon the Update Axiom. Tests $B?$ are interpreted as sets of those pairs $\langle i, i\rangle$ such that B is true in i. Program variables translate themselves; sequencing is translated as relational composition; and choice is interpreted as union. The recursion construct is interpreted as a simultaneous fixpoint. The interpretation of $\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P)$ is obtained by considering all those assignments to $X_1, \ldots, X_n$ for which it is the case that $X_1 = P_1$ and ... and $X_n = P_n$ are all simultaneously true. For each of those assignments P will have a certain value and we get the value of the whole construct by taking the intersection of all values so obtained.

We give the translations of correctness formulas to conclude with. Translations are now of type t and are almost self-explanatory again. The interpretation of the asserted program $\{F_1\}P\{F_2\}$ expresses that after any successful execution of P from a state where F_1 holds we shall be in a state where F_2 holds and $\subseteq$, not unexpectedly, is interpreted as inclusion.

Definition 14 (Translation of Correctness Formulas)

$$(\{F_1\}P\{F_2\})^\dagger = \forall ij\,((F_1^\dagger(i) \wedge P^\dagger(\langle i, j\rangle)) \to F_2^\dagger(j)) \tag{17}$$
$$(P_1 \subseteq P_2)^\dagger = \forall r\,(P_1^\dagger(r) \to P_2^\dagger(r)) \tag{18}$$

7 Scott's Induction Rule

The aim of this section is to import some conceptual machinery from the Scott-Strachey tradition, adapt it to our revised setting, and develop the necessary theory in this new setting. In particular, it will turn out to be possible to prove the validity of *Scott Induction,* a rule which is immensely useful when it comes to proving properties of programs containing the recursion construct.

Our translation function † sends formulae to terms of type $s \to t$, programs to $s \times s \to t$ terms and correctness formulae to terms of type t. Clearly, the domains corresponding to these terms come with natural Boolean algebras: if F is of type $\alpha \to t$ then F is the characteristic function of a set R_F, which may be identified with F itself. This means that $D_{\alpha\to t}$ corresponds to the power set of D_α. The natural Boolean algebra on D_t is also turned into a set algebra if we identify 1 with $\{0\}$, as it is usually done in set theory. Type t and types $\alpha \to t$ will be called *relational* and objects in such domains will be called *relations*. We shall freely talk about unions, intersections, inclusion etc. of relations. In fact, we shall often have occasion to talk about chains $R^0 \subseteq R^1 \subseteq \ldots \subseteq R^n \subseteq \ldots$ of relations of some type, and shall write the union $\bigcup_{n \in \mathcal{N}} R^n$ of such a chain as R^ω. The following definition imports some important notions from the Scott-Strachey tradition into our modeltheoretic setting.

Definition 15 (Continuity, Anticontinuity, Weak anticontinuity) *A term $\mathcal{A}$ of relational type is called (a)* continuous*, (b)* anticontinuous*, (c)* weakly anticontinuous *in the variable ξ of relational type α iff it holds for each model, each assignment a, and each chain of type α relations $R^0 \subseteq R^1 \subseteq \ldots \subseteq R^n \subseteq \ldots$ that*

$$(a)\ \bigcup_n \mathcal{V}(\mathcal{A}, a[R^n/\xi]) = \mathcal{V}(\mathcal{A}, a[R^\omega/\xi])$$

$$(b)\ \bigcap_n \mathcal{V}(\mathcal{A}, a[R^n/\xi]) = \mathcal{V}(\mathcal{A}, a[R^\omega/\xi])$$

$$(c)\ \bigcap_n \mathcal{V}(\mathcal{A}, a[R_n/\xi]) \subseteq \mathcal{V}(\mathcal{A}, a[R^\omega/\xi])$$

It will be our aim to show that translations of programs are continuous in the procedure calls occurring free in them and that translations of correctness formulas are weakly anticontinuous in such procedure calls.

Related to continuity is *monotonicity.* We define the notion below and state the usual lemma which says that it is weaker than continuity.

Definition 16 (Monotonicity) *A term $\mathcal{A}$ of relational type is called* monotonic in ξ *iff $R_1 \subseteq R_2$ implies $\mathcal{V}(\mathcal{A}, a[R_1/\xi]) \subseteq \mathcal{V}(\mathcal{A}, a[R_2/\xi])$.*

Lemma 5 *If $\mathcal{A}$ is continuous in ξ then $\mathcal{A}$ is monotonic in ξ.*

Proof. Consider $R_1 \subseteq R_2 \subseteq R_2 \subseteq \ldots \subseteq R_2 \subseteq \ldots$ $\square$

The next two lemmas generalise the notions of continuity and anticontinuity somewhat. These notions were defined with respect to one variable, but the first lemma shows that that restriction was inessential in the case of continuity while the second lemma proves a generalisation related to weak anticontinuity. Lemma 6 will be applied immediately in the proof of lemma 8 below, but the application of lemma 7 will have to wait a bit, until we come to the proof of Scott's Induction Rule.

Lemma 6 *Let $\mathcal{A}_\alpha$ be continuous in $\xi_1, \ldots, \xi_n$, then*

$$\bigcup_m \mathcal{V}(\mathcal{A}, a[R_1^m/\xi_1, \ldots, R_n^m/\xi_n]) = \mathcal{V}(\mathcal{A}, a[R_1^\omega/\xi_1, \ldots, R_n^\omega/\xi_n])$$

Proof. Repeated application of continuity gives us that

$$\bigcup_{m_1} \cdots \bigcup_{m_n} \mathcal{V}(\mathcal{A}, a[R_1^{m_1}/\xi_1, \ldots, R_n^{m_n}/\xi_n]) = \mathcal{V}(\mathcal{A}, a[R_1^\omega/\xi_1, \ldots, R_n^\omega/\xi_n])$$

That the lefthand side of this equation is equal to

$$\bigcup_m \mathcal{V}(\mathcal{A}, a[R_1^m/\xi_1, \ldots, R_n^m/\xi_n])$$

follows by the monotonicity of $\mathcal{A}$ in each of the variables $\xi_1, \ldots, \xi_n$. $\square$

Lemma 7 *Let $\mathcal{A}_\alpha$ be continuous in $\xi_1, \ldots, \xi_n$, while $\mathcal{B}$ is weakly anticontinuous in ζ_α, and $\xi_1, \ldots, \xi_n$ are not free in $\mathcal{B}$. For each k ($1 \leq k \leq n$), let $R_k^0 \subseteq R_k^1 \subseteq \ldots \subseteq R_k^m \subseteq \ldots$ be a chain, then*

$$\bigcap_m \mathcal{V}([\mathcal{A}/\zeta]\mathcal{B}, a[R_1^m/\xi_1, \ldots, R_n^m/\xi_n]) \subseteq \mathcal{V}([\mathcal{A}/\zeta]\mathcal{B}, a[R_1^\omega/\xi_1, \ldots, R_n^\omega/\xi_n])$$

Proof. For each $m \in \mathcal{N}$ let $S^m = \mathcal{V}(\mathcal{A}, a[R_1^m/\xi_1, \ldots, R_n^m/\xi_n])$. From the monotonicity of $\mathcal{A}$ in each ξ_k we obtain that $S^0 \subseteq S^1 \subseteq \ldots \subseteq S^m \subseteq \ldots$. Hence, by the usual Substitution Theorem and the weak anticontinuity of $\mathcal{B}$ in ζ,

$$\bigcap_m \mathcal{V}([\mathcal{A}/\zeta]\mathcal{B}, a[R_1^m/\xi_1, \ldots, R_n^m/\xi_n]) = \bigcap_m \mathcal{V}(\mathcal{B}, a[S^m/\zeta]) \subseteq \mathcal{V}(\mathcal{B}, a[S^\omega/\zeta])$$

From Lemma 6 it follows that $S^\omega = \mathcal{V}(\mathcal{A}, a\,[R_1^\omega/\xi_1, \ldots, R_n^\omega/\xi_n])$ for each m, so that

$$\mathcal{V}(\mathcal{B}, a[S^\omega/\zeta]) = \mathcal{V}([\mathcal{A}/\zeta]\mathcal{B}, a\,[R_1^\omega/\xi_1, \ldots, R_n^\omega/\xi_n])$$

$\square$

The reason why we are interested in continuity at all is the so-called Knaster-Tarski theorem, which says that intersections of fixpoints as we have used them in the translation of the recursion construct can be 'approximated from below'. The following lemma is a version of the Knaster-Tarski theorem in our modeltheoretic setting. It will be crucial in proving the induction theorem we are after.

Lemma 8 (Knaster-Tarski) *Let $\mathcal{B}$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be terms of type $s \times s \to t$ which are continuous in the type $s \times s \to t$ variables $X_1, \ldots, X_n$. Fix a model and an assignment a. For each k $(1 \le k \le n)$ let $R_k^0 = \emptyset$ and $R_k^{m+1} = \mathcal{V}(\mathcal{A}_k, a[R_1^m/X_1, \ldots, R_n^m/X_n])$. Then $\mathcal{V}(\mathcal{A}_k, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n]) = R_k^\omega$ for each k and*

$$\mathcal{V}(\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \to \mathcal{B}(r)), a) \quad = \quad \mathcal{V}(\mathcal{B}, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n])$$

Proof. First, we check by induction on m that $R_k^0 \subseteq R_k^1 \subseteq \ldots \subseteq R_k^m \subseteq \ldots$, for each k. Trivially, $\emptyset = R_k^0 \subseteq R_k^1$. Suppose that $R_k^m \subseteq R_k^{m+1}$ for each k. Since $\mathcal{A}_k$ is monotonic in $X_1, \ldots, X_n$ it holds that

$$R_k^{m+1} = \mathcal{V}(\mathcal{A}_k, a[R_1^m/X_1, \ldots, R_n^m/X_n]) \subseteq$$

$$\mathcal{V}(\mathcal{A}_k, a[R_1^{m+1}/X_1, \ldots, R_n^{m+1}/X_n]) = R_k^{m+2}$$

which proves the statement. It follows from Lemma 6 that

$$\mathcal{V}(\mathcal{A}_k, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n]) = \bigcup_m \mathcal{V}(\mathcal{A}_k, a[R_1^m/X_1, \ldots, R_n^m/X_n]) =$$

$$\bigcup_m R_k^{m+1} = \bigcup_m R_k^m = R_k^\omega$$

Next, observe that

$$\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \to \mathcal{B}(r))$$

is equivalent with

$$\lambda r.\, \forall Y(\exists X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \wedge Y = \mathcal{B}) \to Y(r))$$

So that the value of this term is the intersection of all S such that $S = \mathcal{V}(\mathcal{B}, a[S_1/X_1, \ldots, S_n/X_n])$ for some $S_1, \ldots, S_n$ which are simultaneous fixpoints in the sense that $S_k = \mathcal{V}(\mathcal{A}_k, a[S_1/X_1, \ldots, S_n/X_n])$ for each k. It follows from the continuity of the terms $\mathcal{A}_k$ that $\mathcal{V}(\mathcal{B}, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n])$ is among those S, so that we have

$$\mathcal{V}(\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \to \mathcal{B}(r)), a) \subseteq \mathcal{V}(\mathcal{B}, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n])$$

To show the converse, suppose that $S_k = \mathcal{V}(\mathcal{A}_k, a[S_1/X_1, \ldots, S_n/X_n])$ for $1 \leq k \leq n$. The monotonicity of the $\mathcal{A}_k$ readily gives us that $R_k^m \subseteq S_k$ for all k and m. It follows that $R_k^\omega \subseteq S_k$ for all k and hence, by $\mathcal{B}$'s monotonicity, that

$$\mathcal{V}(\mathcal{B}, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n]) \subseteq \mathcal{V}(\mathcal{B}, a[S_1/X_1, \ldots, S_n/X_n]).$$

Since $S_1, \ldots, S_n$ were arbitrary simultaneous fixpoints, we conclude that

$$\mathcal{V}(\mathcal{B}, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n]) \subseteq \mathcal{V}(\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \to \mathcal{B}(r)), a)$$

$\square$

Having shown some general properties of continuity and weak anticontinuity to hold, we now turn to proving that the translations we have given in section 6 possess the desired properties. A reader who wants to prove that translations of programs are continuous in the variables X occurring in them will notice that one clause of the required induction cannot be proved using simple continuity properties of union and relational composition. The required extra proof uses lemma 8 and is given below.

Lemma 9 *If $\mathcal{B}$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are terms of type $s \times s \to t$ which are continuous in the type $s \times s \to t$ variables $X_1, \ldots, X_n$ and in ξ_α, then*

$$\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^{n} X_i = \mathcal{A}_i \to \mathcal{B}(r))$$

is also continuous in ξ.

Proof. If ξ is among $\{X_1, \ldots, X_n\}$ we are done, so assume $X_k \neq \xi$. Let $R^0 \subseteq R^1 \subseteq \ldots \subseteq R^n \subseteq \ \ldots$ be an arbitrary chain of type α relations. Define, for each $\ell \in \mathcal{N} \cup \{\omega\}$, each $m \in \mathcal{N}$ and each k $(1 \leq k \leq n)$ the relation $R^m_{\ell,k}$ by setting $R^0_{\ell,k} = \emptyset$ and $R^{m+1}_{\ell,k} = \mathcal{V}(\mathcal{A}_k, a[R^m_{\ell,1}/X_1, \ldots, R^m_{\ell,n}/X_n, R^\ell/\xi])$. Then $R^0_{\ell,k} \subseteq R^1_{\ell,k} \subseteq \ldots \subseteq R^n_{\ell,k} \subseteq \ldots$ for each k and ℓ. Moreover, it is easily proved by induction on m, using the monotonicity of the $\mathcal{A}_k$ and the definition of $R^m_{\ell,k}$ that $R^m_{\ell,k} \subseteq R^m_{\ell+1,k}$ if $\ell \in \mathcal{N}$, so that we also have chains $R^m_{0,k} \subseteq R^m_{1,k} \subseteq \ldots \subseteq R^m_{\ell,k} \subseteq \ldots$. We prove by induction on m that $\bigcup_\ell R^m_{\ell,k} = R^m_{\omega,k}$. This is trivial for $m = 0$. Suppose the statement holds for m. Then, using the continuity of the $\mathcal{A}_k$ and the induction hypothesis, we find that

$$\begin{aligned}
\bigcup_\ell R^{m+1}_{\ell,k} &= \\
\bigcup_\ell \mathcal{V}(\mathcal{A}_k, a[R^m_{\ell,1}/X_1, \ldots, R^m_{\ell,n}/X_n, R^\ell/\xi]) &= \\
\mathcal{V}(\mathcal{A}_k, a\left[\bigcup_\ell R^m_{\ell,1}/X_1, \ldots, \bigcup_\ell R^m_{\ell,n}/X_n, \bigcup_\ell R^\ell/\xi\right]) &= \\
\mathcal{V}(\mathcal{A}_k, a\left[R^m_{\omega,1}/X_1, \ldots, R^m_{\omega,n}/X_n, R^\omega/\xi\right]) &= \\
R^{m+1}_{\omega,k}
\end{aligned}$$

Since $R^\omega_{\omega,k} = \bigcup_m \bigcup_\ell R^m_{\ell,k} = \bigcup_\ell \bigcup_m R^m_{\ell,k}$ it follows that (*) $R^\omega_{\omega,k} = \bigcup_\ell R^\omega_{\ell,k}$. Now we reason as follows, using Lemma 8 twice and using (*) and the continuity of $\mathcal{B}$ in ξ

$$\begin{aligned}
\bigcup_\ell \mathcal{V}(\lambda r.\, \forall X_1 \ldots X_n (\bigwedge_{i=1}^n X_i = \mathcal{A}_i \to \mathcal{B}(r)), a[R^\ell/\xi]) &= \\
\bigcup_\ell \mathcal{V}(\mathcal{B}, a[R^\omega_{\ell,1}/X_1, \ldots, R^\omega_{\ell,n}/X_n, R^\ell/\xi]) &= \\
\mathcal{V}(\mathcal{B}, a\left[\bigcup_\ell R^\omega_{\ell,1}/X_1, \ldots, \bigcup_\ell R^\omega_{\ell,n}/X_n, \bigcup_\ell R^\ell/\xi\right]) &= \\
\mathcal{V}(\mathcal{B}, a\left[R^\omega_{\omega,1}/X_1, \ldots, R^\omega_{\omega,n}/X_n, R^\omega/\xi\right]) &= \\
\mathcal{V}(\lambda r.\, \forall X_1 \ldots X_n (\bigwedge_{i=1}^n X_i = \mathcal{A}_i \to \mathcal{B}(r)), a[R^\omega/\xi])
\end{aligned}$$

This concludes the proof $\square$

That translations of programs are continuous is now easily proved. The proposition will allow us to apply the Knaster-Tarski theorem to those translations.

Proposition 10 *For every program P and procedure call X in P, $P^\dagger$ is continuous in X.*

Proof. By induction on the complexity of P, using the previous lemma for the case that $P = \langle X_i \Leftarrow P_i \rangle_i(Q)$. $\Box$

We can prove a similar connection between correctness formulae and weak anticontinuity. First we note that there are certain syntactic features which will guarantee a type logical term to be continuous or anticontinuous in a given variable.

Lemma 11

1. *If φ is continuous in ξ then $\neg\varphi$ is anticontinuous in ξ;*
2. *If φ is anticontinuous in ξ then $\neg\varphi$ is continuous in ξ;*
3. *If φ, ψ are (anti-)continuous in ξ then $\varphi \wedge \psi$ is (anti-)continuous in ξ;*
4. *If $\mathcal{A}_{\alpha\to t}$ is (anti-)continuous in ξ and ξ does not occur in δ_α then $\mathcal{A}(\delta)$ is (anti-)continuous in ξ;*
5. *If φ is (anti-)continuous in ξ then $\lambda\zeta_\alpha.\varphi$ is (anti-)continuous in ξ;*
6. *If φ is anticontinuous in ξ then $\forall\zeta\,\varphi$ is anticontinuous in ξ;*
7. *If φ is continuous in ξ then $\exists\zeta\,\varphi$ is continuous in ξ.*

Proof. By inspection of the various cases. To prove the first case, for example, assume that φ is continuous in ξ, and let $R^0 \subseteq R^1 \subseteq \ldots \subseteq R^n \subseteq \ldots$ be a chain. Then $\bigcap_n \mathcal{V}(\neg\varphi, a[R^n/\xi]) = 1$ iff, for all n, $\mathcal{V}(\neg\varphi, a[R^n/\xi]) = 1$ iff (by Definition 10), for all n, $\mathcal{V}(\varphi, a[R^n/\xi]) = 0$ iff $\bigcup_n \mathcal{V}(\varphi, a[R^n/\xi]) = 0$ iff (by continuity of φ) $\mathcal{V}(\varphi, a[R^\omega/\xi]) = 0$ iff (by Definition 10) $\mathcal{V}(\neg\varphi, a[R^\omega/\xi]) = 1$.
The other cases are equally simple and are left to the reader. $\Box$

The relation between correctness formulas and weak anticontinuity that was promised is stated and proved as follows.

Proposition 12 *For every correctness formula C and procedure call X in C, $C^\dagger$ is weakly anticontinuous in X.*

Proof. If $C = \{F_1\}P\{F_2\}$ then $C^\dagger = \forall ij.\,(F_1^\dagger(i) \wedge P^\dagger(\langle i,j\rangle)) \rightarrow F_2^\dagger(j)$ and X occurs in P. Using that $P^\dagger$ is continuous in X and applying Lemma 11 repeatedly, we find that $C^\dagger$ is anticontinuous in X. If $C = P_1 \subseteq P_2$ then $C^\dagger = \forall r.\, P_1^\dagger(r) \rightarrow P_2^\dagger(r)$. The following series of equivalences and implications give the required proof.

$$
\begin{array}{rcl}
\bigcap_n \mathcal{V}(\forall r.\, P_1^\dagger(r) \rightarrow P_2^\dagger(r), a[R^n/X]) = 1 & \iff & \\
\text{for all } n\text{: } \mathcal{V}(\forall r.\, P_1^\dagger(r) \rightarrow P_2^\dagger(r), a[R^n/X]) = 1 & \iff & \\
\text{for all } n\text{: } \mathcal{V}(P_1^\dagger, a[R^n/X]) \subseteq \mathcal{V}(P_2^\dagger, a[R^n/X]) & \implies & \\
\bigcup_n \mathcal{V}(P_1^\dagger, a[R^n/X]) \subseteq \bigcup_n \mathcal{V}(P_2^\dagger, a[R^n/X]) & \iff & (P_1,\, P_2 \text{ are continuous}) \\
\mathcal{V}(P_1^\dagger, a[R^\omega/X] \subseteq \mathcal{V}(P_2^\dagger, a[R^\omega/X]) & \iff & \\
\mathcal{V}(\forall r.\, P_1^\dagger(r) \rightarrow P_2^\dagger(r), a[R^\omega/X]) = 1 & &
\end{array}
$$

□

The main rule of Scott Induction can now be proved. We first formulate a general logical variant, from which the Rule itself follows as a corollary.

Theorem 13 *Let $\mathcal{B}$ and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be terms of type $s \times s \rightarrow t$ which are continuous in the type $s \times s \rightarrow t$ variables $X_1, \ldots, X_n$ and let φ be a formula which is weakly anticontinuous in the $s \times s \rightarrow t$ variable X such that $X_1, \ldots, X_n$ are not free in φ. For readability, write $\varphi(\mathcal{A})$ instead of $[\mathcal{A}/X]\varphi$. Then*

$$
\frac{\begin{array}{c}\varphi([\lambda r.\,\texttt{false}/X_1, \ldots, \lambda r.\,\texttt{false}/X_n]\mathcal{B}) \\ \forall X_1 \ldots X_n(\varphi(\mathcal{B}) \rightarrow \varphi([\mathcal{A}_1/X_1, \ldots, \mathcal{A}_n/X_n]\mathcal{B})\end{array}}{\varphi(\lambda r.\, \forall X_1 \ldots X_n(\bigwedge_{i=1}^n X_i = \mathcal{A}_i \rightarrow \mathcal{B}(r)))} AX
$$

Proof. For each k, let $R_k^0 = \emptyset$ and $R_k^{m+1} = \mathcal{V}(\mathcal{A}_k, a[R_1^m/X_1, \ldots, R_n^m/X_n])$. The first premise says that $\mathcal{V}([\mathcal{B}/X]\varphi, a[R_1^0/X_1 \ldots R_n^0/X_n]) = 1$ and the second premise implies that

$$\mathcal{V}([\mathcal{B}/X]\varphi, a[R_1^m/X_1 \ldots R_n^m/X_n]) = 1 \text{ only if}$$

$$\mathcal{V}([\mathcal{B}/X]\varphi, a[R_1^{m+1}/X_1 \ldots R_n^{m+1}/X_n]) = 1.$$

We conclude that $\bigcap_m \mathcal{V}([\mathcal{B}/X]\varphi, a[R_1^m/X_1 \ldots R_n^m/X_n]) = 1$ and use Lemma 7 to derive that $\mathcal{V}([\mathcal{B}/X]\varphi, a[R_1^\omega/X_1 \ldots R_n^\omega/X_n]) = 1$. From this the Theorem follows with Lemma 8. □

Corollary 14 (Scott's Induction Rule) *Assume $X_1, \ldots, X_n$ are not free in C. Write $C(Q)$ for $[Q/X]C$. Then*

$$\frac{\begin{array}{c} C([\mathtt{fail}/X_1, \ldots, \mathtt{fail}/X_n]P)^\dagger \\ \forall X_1 \ldots X_n (C(P)^\dagger \to C([P_1/X_1, \ldots, P_n/X_n]P)^\dagger \end{array}}{C(\langle X_i \Leftarrow P_i \rangle_i (P))^\dagger} AX$$

8 Hoare's Calculus

As a simple application we show that our semantics subsumes the *Hoare Calculus*, which we shall present in Gentzen form. Sequents $\Gamma \vdash_{\mathcal{H}} C$ will consist of a set of asserted programs Γ and an asserted program C. The derivability relation $\vdash_{PA}$ between assertions F is given by an axiomatization of first-order logic plus the first-order Peano axioms. Structural rules of our Hoare-Gentzen calculus are the following.

$$C \vdash_{\mathcal{H}} C \qquad \text{(Id)}$$

$$\frac{\Gamma_1 \vdash_{\mathcal{H}} C}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{H}} C} \qquad \text{(Weakening)}$$

$$\frac{\Gamma_1 \vdash_{\mathcal{H}} C_1 \quad \Gamma_2, C_1 \vdash_{\mathcal{H}} C_2}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{H}} C_2} \qquad \text{(Cut)}$$

The following logical rules embody the Hoare Calculus.

$$\vdash_{\mathcal{H}} \{[N/x]F\} x := N \{F\} \qquad \text{(Assignment)}$$

$$\vdash_{\mathcal{H}} \{F\} B? \{F \wedge B\} \qquad \text{(Test)}$$

$$\frac{\Gamma_1 \vdash_{\mathcal{H}} \{F_1\} P_1 \{F_2\} \quad \Gamma_2 \vdash_{\mathcal{H}} \{F_2\} P_2 \{F_3\}}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{H}} \{F_1\} P_1; P_2 \{F_3\}} \qquad \text{(Composition)}$$

$$\frac{\Gamma_1 \vdash_{\mathcal{H}} \{F_1\} P_1 \{F_2\} \quad \Gamma_2 \vdash_{\mathcal{H}} \{F_1\} P_2 \{F_2\}}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{H}} \{F_1\} P_1 \cup P_2 \{F_2\}} \qquad \text{(Choice)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathcal{H}} \{F_1\}[\mathtt{fail}/X_1, \ldots, \mathtt{fail}/X_n]P\{F_2\} \\ \Gamma, \{F_1\}P\{F_2\} \vdash_{\mathcal{H}} \{F_1\}[P_1/X_1, \ldots, P_n/X_n]P\{F_2\} \end{array}}{\Gamma \vdash_{\mathcal{H}} \{F_1\} \langle X_i \Leftarrow P_i \rangle_{i=1}^n (P) \{F_2\}} \qquad \text{(Recursion)}$$

$$\frac{F_1 \vdash_{PA} F_2 \quad \Gamma \vdash_{\mathcal{H}} \{F_2\} P \{F_3\} \quad F_3 \vdash_{PA} F_4}{\Gamma \vdash_{\mathcal{H}} \{F_1\} P \{F_4\}} \qquad \text{(Consequence)}$$

Example 2 As an example of an (informal) derivation in this calculus, we show that

$$\frac{\Gamma, \{F_1\}X\{F_2\} \vdash_{\mathcal{H}} \{F_1\}P\{F_2\}}{\Gamma \vdash_{\mathcal{H}} \{F_1\}\mu X(P)\{F_2\}} \qquad (\mu)$$

is a derived rule:

1. $\vdash_{\mathcal{H}} \{F_1\}\mathtt{fail}\{F_1 \wedge \mathtt{false}\}$ (Test)
2. $\vdash_{\mathcal{H}} \{F_1\}\mathtt{fail}\{F_2\}$ (Consequence, 1)
3. $\Gamma \vdash_{\mathcal{H}} \{F_1\}\mathtt{fail}\{F_2\}$ (Weakening, 2)
4. $\Gamma, \{F_1\}X\{F_2\} \vdash_{\mathcal{H}} \{F_1\}P\{F_2\}$ (premise)
5. $\Gamma \vdash_{\mathcal{H}} \{F_1\}\langle X \Leftarrow P\rangle(X)\{F_2\}$ (Recursion, 3,4)

Example 3 A second example uses the rule we have just derived to show that

$$\frac{\Gamma \vdash \{F \wedge B\}P\{F\}}{\Gamma \vdash \{F\}\mathtt{while}\ B\ \mathtt{do}\ P\ \mathtt{od}\ \{F \wedge \neg B\}} \qquad (\text{while})$$

is another derived rule.

1. $\Gamma \vdash_{\mathcal{H}} \{F \wedge B\}P\{F\}$ (premise)
2. $\vdash_{\mathcal{H}} \{F\}B?\{F \wedge B\}$ (Test)
3. $\Gamma \vdash_{\mathcal{H}} \{F\}B?; P\{F\}$ (Composition, 1,2)
4. $\{F\}X\{F \wedge \neg B\} \vdash_{\mathcal{H}} \{F\}X\{F \wedge \neg B\}$ (Id)
5. $\Gamma, \{F\}X\{F \wedge \neg B\} \vdash_{\mathcal{H}} \{F\}B?; P; X\{F \wedge \neg B\}$ (Composition, 3,4)
6. $\vdash_{\mathcal{H}} \{F\}\neg B?\{F \wedge \neg B\}$ (Test)
7. $\Gamma, \{F\}X\{F \wedge \neg B\} \vdash_{\mathcal{H}} \{F\}(B?; P; X) \cup \neg B?\{F \wedge \neg B\}$ (Choice, 5,6)
8. $\Gamma \vdash_{\mathcal{H}} \{F\}\mu X((B?; P; X) \cup \neg B?)\{F \wedge \neg B\}$ (μ, 7)

We can give a rather straightforward proof of the soundness of the calculus $\mathcal{H}$ by simply considering the translations of its axioms and rules. For any set of asserted programs Γ, we write $\Gamma^\dagger$ for the set of translations $\{C^\dagger \mid C \in \Gamma\}$.

Theorem 15 (Soundness of $\mathcal{H}$) $\Gamma \vdash_{\mathcal{H}} C$ *implies* $\Gamma^\dagger \models_{AX} C^\dagger$

Proof. We need to check that axioms and rules of the Hoare-Gentzen Calculus continue to hold if we replace every $\vdash_{\mathcal{H}}$ by $\models_{AX}$ and every asserted program C by its translation $C^\dagger$. For example, Composition is translated to:

$$\frac{\begin{array}{c}\Gamma_1^\dagger \models_{AX} \forall ij((F_1^\dagger(i) \wedge P_1^\dagger(\langle i, j\rangle)) \to F_2^\dagger(j)) \\ \Gamma_2^\dagger \models_{AX} \forall ij((F_2^\dagger(i) \wedge P_2^\dagger(\langle i, j\rangle)) \to F_3^\dagger(j))\end{array}}{\Gamma_1^\dagger, \Gamma_2^\dagger \models_{AX} \forall ij((F_1^\dagger(i) \wedge \exists k(P_1^\dagger(\langle i, k\rangle) \wedge P_2^\dagger(\langle k, j\rangle))) \to F_3^\dagger(j))}$$

This is easily seen to be valid on the force of predicate logic alone. Similarly, the structural rules, the Test axiom, Choice and Consequence lead to translations whose validity is easily verified. The soundness of the Recursion rule follows almost directly from Scott Induction.

This leaves it for us to verify the soundness of the Assignment axiom. We must check that

$$\models_{AX} \forall ij((([N/x_k]F)^\dagger(i) \wedge i[S^k(0)]j \wedge V(S^k(0))(j) = N^\dagger(i)) \to F^\dagger(j))$$

Using Lemma 4 and the notation that was defined in that lemma, we find that this is equivalent with

$$\models_{AX} \forall ij((([N/x_k]F)^i \wedge i[S^k(0)]j \wedge V(S^k(0))(j) = N^i) \to F^j)$$

Letting F' be the result of substituting $V(S^n(0))(i)$ for x_n in F, for each $n \neq k$, we see that the latter is equivalent with

$$\models_{AX} \forall ij((([N^i/x_k]F') \wedge i[S^k(0)]j \wedge V(S^k(0))(j) = N^i) \to F^j)$$

in its turn. But this last statement is easily seen to hold on the basis of the definition of $i[S^k(0)]j$. $\square$

9 The Union Theorem

The main theorem of this section, and indeed of the paper, is Theorem 23 below, which states that the meaning of a program is equal to the union of the meanings of all linear programs deriving it. Before we prove the theorem itself, it will be expedient to consider a series of lemmas and a proposition. The lemma and the proposition below establish the soundness of the calculus $\mathcal{C}$ and are therefore of interest in themselves.

Lemma 16 (Soundness of the ρ rule)

$$\mathcal{V}((\langle X_i \Leftarrow P_i \rangle_i (P))^\dagger, a) = \mathcal{V}(([\langle X_i \Leftarrow P_i \rangle_i (P_k)/X_k]_{k=1}^n P)^\dagger, a)$$

Proof. Directly from Lemma 8 and Proposition 10. □

Proposition 17 (Soundness of $\mathcal{C}$) $P_1 \vdash_{\mathcal{C}} P_2 \Rightarrow \mathcal{V}(P_1^\dagger, a) \subseteq \mathcal{V}(P_2^\dagger, a)$

Proof. The proof proceeds by induction on the length of the derivation of P_2 from P_1. □

We now come to a series of five lemmas which are of a more technical nature. They are needed for the proof of Theorem 23 below. Let us begin with three simple lemmas of a proof-theoretical nature. Lemma 18 says that a program P can only derive a sequence $P_1; P_2$ if P can be split into two parts which derive P_1 and P_2 respectively; lemma 20 establishes a useful substitution property and uses lemma 19 for its proof.

Lemma 18 *$P \vdash_{\mathcal{C}} P_1; P_2$ iff there are P_1' and P_2' such that $P = P_1'; P_2'$, $P_1' \vdash_{\mathcal{C}} P_1$ and $P_2' \vdash_{\mathcal{C}} P_2$.*

Proof. A straightforward induction on length of derivation. □

Lemma 19 *Let i and ℓ range over $\{1, \ldots, n\}$ and let k range over $\{1, \ldots, m\}$, then*

1. *$[P_k/X_k]_k (Q_1 \cup Q_2)$ follows from $[P_k/X_k]_k Q_1$ and from $[P_k/X_k]_k Q_2$ by the $\cup$ rule;*

2. *$[P_k/X_k]_k \langle Y_i \Leftarrow Q_i \rangle_i (Q)$ follows from $[P_k/X_k]_k [\langle Y_i \Leftarrow Q_i \rangle_i (Q_\ell)/Y_\ell]_\ell Q$ by the ρ rule.*

Proof. The first statement is trivial, so we prove the second. Let $Z_1, \ldots, Z_m$ be the first variables in some given ordering which do not occur in any of the P_k, in $\langle Y_i \Leftarrow Q_i \rangle_i (Q)$, or in any of the X_k. For any P, abbreviate $[P_k/Y_k]_k [Z_\ell/Y_\ell]_\ell P$ as P'. We have

$$\begin{array}{rcl}
[P_k/X_k]_k [\langle Y_i \Leftarrow Q_i \rangle_i (Q_\ell)/Y_\ell]_\ell Q & = & \\
[P_k/X_k]_k [\langle Y_i \Leftarrow Q_i \rangle_i (Q_\ell)/Z_\ell]_\ell [Z_\ell/Y_\ell]_\ell Q & = & \text{by Lemma 1} \\
[[P_k/X_k]_k \langle Y_i \Leftarrow Q_i \rangle_i (Q_\ell)/Z_\ell]_\ell Q' & = & \text{by Definition 3} \\
[\langle Z_i \Leftarrow Q_i' \rangle_i (Q_\ell')/Z_\ell]_\ell Q' & &
\end{array}$$

This means that $[P_k/X_k]_k \langle Y_i \Leftarrow Q_i \rangle_i (Q)$, which by definition is $\langle Z_i \Leftarrow Q_i' \rangle_i (Q')$, follows from $[P_k/X_k]_k [\langle Y_i \Leftarrow Q_i \rangle_i (Q_\ell)/Y_\ell]_\ell Q$ by the ρ rule. □

Lemma 20 *If* $Q \vdash_{\mathcal{C}} Q'$ *then* $[P_k/X_k]_k\, Q \vdash_{\mathcal{C}} [P_k/X_k]_k\, Q'$.

Proof. Let $Q_1, \ldots, Q_n$ be a derivation of Q' from Q. It follows immediately from the previous lemma that, for $1 \leq i \leq n$, $[P_k/X_k]_k\, Q_{i+1}$ is an immediate consequence of $[P_k/X_k]_k\, Q_i$. Hence $[P_k/X_k]_k\, Q_1, \ldots, [P_k/X_k]_k\, Q_n$ is a derivation of $[P_k/X_k]_k\, Q'$ from $[P_k/X_k]_k\, Q$. $\Box$

The next two lemmas establish properties of the union of the denotations of all linear programs deriving a given program.

Lemma 21 *If* $R_1 \subseteq \bigcup\{\mathcal{V}(L_1^\dagger, a) \mid L_1 \vdash_{\mathcal{C}} Q_1\}$ *and* $R_2 \subseteq \bigcup\{\mathcal{V}(L_2^\dagger, a) \mid L_2 \vdash_{\mathcal{C}} Q_2\}$ *then* $R_1 \circ R_2 \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} Q_1; Q_2\}$

Proof. From the assumptions and the translation of $L_1; L_2$ it follows that

$$\begin{aligned} R_1 \circ R_2 &\subseteq \bigcup\{\mathcal{V}(L_1^\dagger, a) \circ \mathcal{V}(L_1^\dagger, a) \mid L_1 \vdash_{\mathcal{C}} Q_1, L_1 \vdash_{\mathcal{C}} Q_2\} \\ &= \bigcup\{\mathcal{V}((L_1; L_2)^\dagger, a) \mid L_1 \vdash_{\mathcal{C}} Q_1, L_1 \vdash_{\mathcal{C}} Q_2\} \end{aligned}$$

But since Lemma 18 tells us that $L \vdash_{\mathcal{C}} Q_1; Q_2$ iff there are L_1 and L_2 such that $L = L_1; L_2$ while $L_1 \vdash_{\mathcal{C}} Q_1$ and $L_2 \vdash_{\mathcal{C}} Q_2$, the latter term is equal to $\bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} Q_1; Q_2\}$. $\Box$

Lemma 22 *Suppose* $R_k \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} Q_k\}$ *for all* k $(1 \leq k \leq n)$. *Then for all linear programs L:*

$$\mathcal{V}(L^\dagger, a[R_1/X_1, \ldots, R_n/X_n]) \subseteq \bigcup\{\mathcal{V}(L'^\dagger, a) \mid L' \vdash_{\mathcal{C}} [Q_1/X_1, \ldots, Q_n/X_n]L\}$$

Proof. This follows by induction on the number m of occurrences of the variables $X_1, \ldots, X_n$ in L. If $m = 0$ the statement reduces to $\mathcal{V}(L^\dagger, a) \subseteq \bigcup\{\mathcal{V}(L'^\dagger, a) \mid L' \vdash_{\mathcal{C}} L\}$, which is obviously true. If $m > 0$ then L has the form $L_1; X_k; L_2$, with fewer than m occurrences of $X_1, \ldots, X_n$ in L_1 and L_2. Write a' for $a[R_1/X_1, \ldots, R_n/X_n]$. Then

$$\mathcal{V}(L^\dagger, a') = \mathcal{V}(L_1^\dagger, a') \circ R_k \circ \mathcal{V}(L_2^\dagger, a').$$

From the induction hypothesis it follows that

$$\mathcal{V}(L_i^\dagger, a') \subseteq \bigcup\{\mathcal{V}(L'^\dagger, a) \mid L' \vdash_{\mathcal{C}} [Q_1/X_1, \ldots, Q_n/X_n]L_i\}$$

for $i \in \{1, 2\}$. This, together with the assumption $R_k \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} Q_k\}$ and Lemma 21 allows us to draw the desired conclusion that

$$\mathcal{V}(L^\dagger, a') \subseteq \bigcup\{\mathcal{V}(L'^\dagger, a) \mid L' \vdash_{\mathcal{C}} [Q_1/X_1, \ldots, Q_n/X_n]L_1; X_k; L_2\}$$

$\Box$

The Union Theorem can now be proved. It was already clear that execution of some program P must consist in execution of some L such that $L \vdash_{\mathcal{C}} P$ and, conversely, that execution of any such L amounts to execution of P. The theorem shows that this squares with the modeltheoretic interpretation of programs which we have obtained by our translation and in this way provides a justification for that translation.

Theorem 23 (Union Theorem) *$\mathcal{V}^{\mathcal{M}}(P^\dagger, a) = \bigcup\{\mathcal{V}^{\mathcal{M}}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} P\}$ for all models $\mathcal{M}$ and assignments a.*

Proof. That $\bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} P\} \subseteq \mathcal{V}(P^\dagger, a)$ follows from Proposition 17. We prove the reverse, the statement that $\mathcal{V}(P^\dagger, a) \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} P\}$, by induction on the complexity of P. In case P is atomic the only L such that $L \vdash_{\mathcal{C}} P$ is P itself, so the statement is trivially true. In case P is $P_1 \cup P_2$ or P is $P_1; P_2$ and the statement holds for P_1 and P_2, we can easily derive the statement for P itself. This leaves it for us to prove that

$$\mathcal{V}(((\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_{n+1}))^\dagger, a) \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_{n+1})\}$$

under the induction hypothesis (*) that for all k $(1 \leq k \leq n+1)$ $\mathcal{V}(P_k^\dagger, a) \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} P_k\}$ for all assignments a. In fact, we shall strengthen the statement somewhat and use the induction hypothesis to prove that

$$\mathcal{V}(((\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k))^\dagger, a) \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)\}$$

for all k $(1 \leq k \leq n+1)$. To this end, define relations R_k^m for all $m \in \mathcal{N}$ by setting $R_k^0 = \emptyset$ and $R_k^{m+1} = \mathcal{V}(P_k^\dagger, a[R_1^m/X_1, \ldots, R_n^m/X_n])$. Using a subinduction on m we prove that $R_k^m \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)\}$ for all m and k $(1 \leq k \leq n+1)$. Since this is trivially true for $m = 0$, suppose (**) the statement holds for m. The induction hypothesis (*) gives us that

$$\begin{aligned} R_k^{m+1} &= \mathcal{V}(P_k^\dagger, a[R_1^m/X_1, \ldots, R_n^m/X_n]) \\ &\subseteq \bigcup\{\mathcal{V}(L'^\dagger, a[R_1^m/X_1, \ldots, R_n^m/X_n]) \mid L' \vdash_{\mathcal{C}} P_k\} \end{aligned}$$

On the other hand, Lemma 22 in combination with the subinduction hypothesis (**) tells us that, for any linear program L':

$$\mathcal{V}(L'^\dagger, a[R_1^m/X_1, \ldots, R_n^m/X_n]) \subseteq$$

$$\bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} [\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_j)/X_j]_{j=1}^n L'\}$$

Combining our findings, we conclude that

$$R_k^{m+1} \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid \exists L' : L \vdash_{\mathcal{C}} [\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_j)/X_j]_{j=1}^n L' \& L' \vdash_{\mathcal{C}} P_k\}$$

Since, by Lemma 20, $L' \vdash_{\mathcal{C}} P_k$ implies that

$$[\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_j)/X_j]_{j=1}^n L' \vdash_{\mathcal{C}} [\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_j)/X_j]_{j=1}^n P_k$$

It follows that

$$\begin{array}{rcl} R_k^{m+1} & \subseteq & \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} [\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_j)/X_j]_{j=1}^n P_k\} \\ & \subseteq & \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)\} \end{array}$$

We may conclude that $\bigcup_m R_k^m \subseteq \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)\}$ for all k and now reason as follows.

$$\begin{array}{rcl} \mathcal{V}((\langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k))^\dagger, a) & = & \text{(Lemma 8)} \\ \mathcal{V}(P_k^\dagger, a[R_1^\omega/X_1, \ldots, R_n^\omega/X_n]) & = & \text{(continuity)} \\ \bigcup_m \mathcal{V}(P_k^\dagger, a[R_1^m/X_1, \ldots, R_n^m/X_n]) = \bigcup_m R_k^m & \subseteq & \\ \bigcup\{\mathcal{V}(L^\dagger, a) \mid L \vdash_{\mathcal{C}} \langle X_i \Leftarrow P_i\rangle_{i=1}^n(P_k)\} & & \end{array}$$

This concludes the proof. $\Box$

10 Conclusion and Further Prospects

In this paper we have shown that at least some non-trivial programming constructs, including recursion, can be provided with a semantics in the models of ordinary second-order logic, using the technique of translating programs and correctness statements into that logic as it is commonly done in Montague Semantics. The programming language which we have treated was very small of course, but there seem to be no reasons why we should not be able in principle to extend it with many useful constructs. This is one way to continue from here, but there are other possibilities as well. We speculate on two of them.

- In this paper we have only looked at an imperative language, but the procedural semantics of *logic programming* languages is usually done in the denotational framework as well and makes extensive use of fixpoint constructions. The idea that we could have a *logical* treatment of the procedural semantics of these languages is tantalising.

- We have taken the naive view that the meaning of a program consists in a relation between input and output states. But more finegrained notions of meaning (for example, the meaning of a program as a *process* or as an *execution trace*) do not seem to be in conflict with the methods employed here. As long as we let programs be translated as terms of relational type the methods of section 7 seem to be applicable.

References

[1] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Academic Press, Orlando, Florida, 1975.

[2] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] J. De Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[4] D. Gallin. *Intensional and Higher-Order Modal Logic.* North-Holland, Amsterdam, 1975.

[5] R. Goldblatt. *Logics of Time and Computation.* CSLI Lecture Notes, Stanford, 1987.

[6] J. Groenendijk and M. Stokhof. Dynamic Montague Grammar. In L. Kálmán and L. Pólos, editors, *Papers from the Second Symposium on Logic and Language*, pages 3–48. Akadémiai Kiadó, Budapest, 1990.

[7] D. Harel. Dynamic Logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II, pages 497–604. Reidel, Dordrecht, 1984.

[8] L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[9] L. Henkin. A Theory of Propositional Types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[10] Th. Janssen. *Foundations and Applications of Montague Grammar.* Centre for Mathematics and Computer Science, Amsterdam, 1986.

[11] R. Montague. Universal Grammar. In *Formal Philosophy*, pages 222–246. Yale University Press, New Haven, 1970.

[12] R. Montague. The Proper Treatment of Quantification in Ordinary English. In *Formal Philosophy*, pages 247–270. Yale University Press, New Haven, 1973.

[13] P. Mosses. Denotational Semantics. In A.J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 577–632. Elsevier, Amsterdam, 1990.

[14] R.A. Muskens. Anaphora and the Logic of Change. In J. Van Eijck, editor, *Logics in AI, Proceedings of JELIA '90*. Springer-Verlag, Berlin, 1991.

[15] R.A. Muskens. Tense and the Logic of Change. In U. Egli, E.P. Pause, C. Schwarze, A. Von Stechow, and G. Wienold, editors, *Lexical Knowledge in the Organization of Language*, pages 147–183. John Benjamins, Amsterdam, 1995.

[16] R.A. Muskens. Combining Montague Semantics and Discourse Representation. *Linguistics and Philosophy*, 19:143–186, 1996.

[17] R.A. Muskens, J.F.A.K. van Benthem, and A. Visser. Dynamics. In J.F.A.K. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 587–648. Elsevier, Amsterdam, 1997.

[18] V.R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *Proc. 17th IEEE Symp. Found. Comp. Sci.*, pages 46–57, 1976.

[19] D.S. Scott. Domains for Denotational Semantics. In M. Nielsen and E. Schmidt, editors, *Proceedings 9th International Colloquium on Automata, Languages and Programming*, pages 577–613, 1982.

[20] D.S. Scott and Strachey C. Toward a Mathematical Semantics for Computer Languages. In *Proc. Symp. on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971.

[21] D.S. Scott and J.W. De Bakker. A Theory of Programs, 1969. Unpublished Notes. IBM Vienna.

[22] R.D. Tennent. Denotational Semantics. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 169–322. Clarendon Press, Oxford, 1992.

[23] J.F.A.K van Benthem. Modal State Semantics, 1994. Unpublished Manuscript.

[24] J.F.A.K. van Benthem. *Exploring Logical Dynamics.* CSLI, Stanford, 1996.

[25] J.F.A.K. van Benthem and K. Doets. Higher-Order Logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume I, pages 275–329. Reidel, Dordrecht, 1983.

[26] J. van Eijck. Typed Logics with States. *Journal of the IGPL*, 1997.