A Typed Operational Semantics Based on Grammatical Characterisation of an Abstract Machine

Robin Popplestone
Visiting from The Department of Computer Science
University of Massachusetts at Amherst

Abstract

In computation, we often want to represent finite sequences of objects. Classically, a set of finite sequences of *tokens* drawn from an *alphabet* is a *language*, and may be characterised by a *grammar*. It would seem natural therefore to use grammars as a richer way of specifying types, simply by deciding that *tokens* can be the *objects* from which a type is built . For example, `List({Int*String})` is the type of all lists in which short integers alternate with strings, using $\{a\}$ for the Kleene-closure of a.

However, one's enthusiasm for this approach might be tempered by the realisation that some questions about some grammars are not effectively computable. A crucial capability required for this work is the ability to *divide* one grammar *exactly* by another. Exact division is an extension of the concept of division found in Hopcroft and Ullman and characterises, at the type-level of abstraction, the acquisition of arguments by a function.

This approach is valuable for any language or system in which stacking operations are explicit, including the Forth, Pop-11 and PostScript languages, and the Poplog multi-language environment. It has been applied experimentally to the development of a type-checker for the POP-11 language.

1 Introduction

Many of the structures occurring in computation can be regarded as representations of finite sequences of objects. Typically therefore a program will contain many variables whose legal bindings are drawn from a set of finite sequences. Classically, a set of finite sequences of *tokens* drawn from an *alphabet* is a *language*, and may be characterised by a *grammar*. It would seem natural therefore to use grammars as a richer way of specifying the types of variables, simply by deciding that *tokens* can be *objects* of the language. For example, I use `List(Int*String)` as the type of all lists of two members, the first of which is a (short) integer, and the second of which is a string. Likewise `List({Int*String})` is the type of all lists in which short integers alternate with strings.

Concatenation of such sequences is a natural operation, leading to the idea that the product operation on types should be *associative*. The grammatical approach supports this associativity, whereas more standard approaches to type in functional languages do not. Of course, associativity is purchased at a cost, since the existence of a most general unifier, which is the basis of the derivation of principal types, cannot be guaranteed.

Moreover, one's enthusiasm for the grammatical approach might be tempered by the realisation that some questions about some grammars are not effectively computable. For example, we may wish to know if two types are identical. These considerations have led me to restrict myself to building on a basis of *regular grammars*, although the requirements of parametric polymorphism have mean that using what might be called *regular algebras* is more appropriate.

I have built an experimental grammatical type-checker for the POP-11 language. POP-11, with its *open stack* has always been regarded as not capable of being statically type-checked, but it has proved possible to create a type-checker which, though it restricts the programmer, does so in the direction of requiring that (s)he adopts recognised good practice, rather than being irksome. It is based on the idea of maintaining a *grammar* which characterises the state of the *stack* at each step in program execution, as specified in the Poplog Virtual Machine.

A crucial capability required for this work is the ability to *divide* one grammar *exactly* by another. Exact division is an extension of the concept of division found in Hopcroft and Ullman. Essentially, at the type level of abstraction, a function *right-divides* the stack-grammar by its *argument grammar*, and then multiplies by its *result grammar*. Left-division is also required, for example the *tl* function has type:

```
tl: List of a -> List of Tl(a);
```

where `Tl` is a grammar-function which yields the left-quotient of a grammar divided by the *top monolog*, and a is a type-variable.

A number of other computer languages, including Forth and PostScript, exist in which the method of passing parameters to a function or procedure is explicitly defined to be a stack. The methods discussed in this paper are relevant to type-checking programs in these languages, and has been tested on a Forth subset.

A stack-state can be regarded as a *sentence* in a *language* characterised by a *grammar*. Each different point in a program thus determines a set of stack-states, that is to say a language. The *alphabet* of this language is the set of objects of the programming language. Within this framework, the possible values of a *variable* are a language all of whose sentences are of length exactly one. Pushing a variable on the stack corresponds to taking the *product* of the stack-language and the variable-language; popping off the stack into a variable corresponds to taking the *exact quotient* of the stack-language by the variable-language. The call of a procedure can be characterised as taking the *exact quotient* of a stack-language by the argument-language for the procedure, and then multiplying the result by the result-language for the procedure.

Control constructs in a language can be treated by introducing a new non-terminal symbol to denote the stack-state at a program-point where a confluence can occur — this will be a statement-label in the lowest level of language. Each transfer of control to this point generates a *production* relating the non-terminal to the stack-grammar for the point from which control was transferred. This gives rise to context-free grammars characterising the behaviour of a code-sequence (normally a procedure-body). The existing type-checker converts linear grammars into regular expressions. Non-linear grammars are regarded as type-errors.

1.1 A guide to the paper

Following this introduction, the paper is divided into the following major sections:

- Languages: sums, products and quotients. This is a treatment of the basic properties of formal languages as found in [4], but with an emphasis on the quotient operation required for the type-checker.
- The Virtual Machine: This introduces the idealised $(s, \mathbf{e}, c, b, h)$ machine in which instruction b_c in a code-block b, taking arguments from a stack s acts on a heap h, interpreting variables as specified by the environment $\mathbf{e}$. We define the *exec* function which executes one instruction, and the *obey* function which obeys a complete code-block.
- Grammars characterise languages: Languages are treated abstractly, as sets of sequences, in the section above. In this section we introduce Extended Regular Word Algebras (ERWA's), which, together with a type-environment, define languages composed of objects in a given machine state. We prove some *stability* lemmas, which show how these languages vary as parameters such as machine-state vary.
- Describing Machine States: In this section we first define a formalism for characterising a machine-state using an element of an ERWA together with a *term* to specify the stack and a type-environment to specify the machine-environment. We then define an *annotation* of a code-block, and show that an annotation is a correct description of the behaviour of a machine.

2 Languages: sums, products and quotients

In this section we remind readers of the basic properties of languages, and develop some propositions which are intended primarily to support the simplification of grammars which characterise languages.

There is a considerable advantage in developing our theory of type with *languages* rather than the *grammars* which characterise them since it allows us to identify the operations we need to perform without being tied to any particular grammatical apparatus. Consequently we are able to develop an approach to describing associative types which allows us choice of descriptive apparatus. In particular, while the operations on languages that we introduce will be those of regular grammars, we will our grammars, which may contain type-variables, are not therefore regular.

For definitions of operations on languages see [4]

We shall use the symbol '$\mathbf{1}$' for the language which consists of the empty string ϵ. This corresponds to the `unit` type of SML.

Definition 2.0.1 (Language) *Let Σ be an* alphabet *of* objects. *A language L is a set of sequences of members of Σ. We write $\Sigma_0(L)$ for the set of objects that actually occur in a language L.*

We denote the empty language by $\emptyset$.

Definition 2.0.2 (Monologs) *A language L is said to be a* monolog *if it is non-empty and contains only sequences of length exactly 1.*

We will use the capital letters K, J, L for languages, M, N for *monolog* languages. If $a \in \Sigma$ is an object, then the sequence of length 1 consisting of a will be written $\hat{a}$.

Definition 2.0.3 (Singletons) *If $\sigma \in \Sigma$ then the language $S(\sigma) = \{\hat{\sigma}\}$ is called a singleton.*

Definition 2.0.4 (Monotonicity) *Let $\mathcal{F}$ be a function on languages. We say that $\mathcal{F}$ is* monotonic *on argument i if $L_i \subseteq L_i'$ implies $\mathcal{F}(L_1, \ldots, L_{i-1}, L_i, L_{i+1} \ldots L_n) \subseteq \mathcal{F}(L_1, \ldots, L_{i-1}, L_i', L_{i+1} \ldots L_n)$. We say that it is monotonic if it is monotonic on all arguments*

2.1 The product and union of Languages

Languages, being sets, have the normal operations of boolean algebra defined over them. We write $L \subseteq K$ if every member of L is a member of K. Note that this includes the possiblity that the sets are equal. There is a top element $\top$. The union of languages is of more interest to us than the intersection. There is also a *top monolog* $\top_M$.

The *union* of languages is the standard set-theoretic union:

$$L \cup K = \{l | l \in L \text{ or } l \in K\}$$

The algebraic laws governing union and intersection are of course just those of boolean algebra, including:

$$L \cup K = K \cup L, \ (J \cup K) \cup L = J \cup (K \cup L)$$

The concatenation operation on sequences gives rise to the *product* of languages.

Definition 2.1.1 *If L and K are languages, then their* product *is:*

$$LK = \{lk | l \in L, k \in K\}$$

The product of languages is *associative* and distributes over union. It is not commutative.

$$J(KL) = (JK)L \ (J \cup K)L = JL \cup KL, \ L(J \cup K) = LJ \cup LK$$

$\mathbf{1}$ acts as an identity, so that $\mathbf{1}L = L = L\mathbf{1}$ for any language L.

For any integer $n \geq 0$, we define L^n by $L^0 = \mathbf{1}$, $L^n = LL^{n-1}$. Thus $L^1 = L$.

The operations of union, intersection and product of languages are monotonic. Complementation is not.

2.2 The Kleene Closure

Definition 2.2.1 (Kleene Closure) *If L is a language, then*

$$\{L\} = \bigcup_{i=0}^{\infty} L^i$$

is called the Kleene-closure of L. We need a modified version of this operation, where the first power in the union is L^n, and write:

$$\{L\}_n = \bigcup_{i=n}^{\infty} L^i$$

.

¿From the definition the following lemmas are immediate:

Lemma 2.2.1 $K = \mathbf{1} \cup K\{K\} = \mathbf{1} \cup \{K\}K$

Lemma 2.2.2 *Kleene closure is monotonic*

Definition 2.2.2 (Regular languages) *If Σ is an alphabet, then any language L which is formed by taking $\mathbf{1}$, $\emptyset$, singletons $S(\sigma)$ where $\sigma \in \Sigma$, combined by union, product, Kleene-closure is said to be* regular.

Proposition 2.2.1 *For any language L, $\{\{L\}_n\}_m = \{L\}_{mn}$*

Proof: Let $l \in \{\{L\}_n\}_m$ Then $l = l_1 \ldots l_{m'}$, $m' \geq m$ where each $l_i = l_{i1} \ldots l_{in_i}$, $n_i > n$, $l_{ij} \in L$, So $l = (l_{11} \ldots l_{1n_1}) \ldots (l_{m1} \ldots l_{mn_m})$ Hence, by associativity, given that there are at least mn terms in the product, $l \in \{L\}_{mn}$

Conversely, let $l \in \{L\}_{mn}$. Then $l = l_1 \ldots l_k$, $k \geq mn$ where each $l_i \in L$. By associativity, we can write this as
$l = (l_1 \ldots l_m)(l_{m+1} \ldots l_{2m}) \ldots (l_{m(n-1)} \ldots l_{mn} l_{mn+1} \ldots l_k) \in \{\{L\}_n\}_m$

Proposition 2.2.2 *If $L_1 \ldots L_n$ are languages, then the product*

$$L_n\{L_1 L_2 \ldots L_n\}_m = \{L_n L_1 \ldots L_{n-1}\}_m L_n$$

Proof:
Let $l \in L_n\{L_1L_2 \dots L_n\}_m$ Then

$$l = l_n(l_{11}l_{21} \dots l_{n1})(l_{12}l_{22} \dots l_{n2}) \dots (l_{1m'}l_{2m'} \dots l_{nm'})$$

where $l_{ij} \in L_i, m' \geq m$. Now associativity allows us to rebracket:

$$l = (l_nl_{11}l_{21} \dots l_{(n-1)1})(l_{n1}l_{12}l_{22} \dots l_{(n-1)2}) \dots (l_{n(m'-1)}l_{1m'}l_{2m'} \dots l_{n-1m'})l_{nm'}$$

Hence $l \in \{L_nL_1 \dots L_{n-1}\}_mL_n$
The converse membership is proved analogously.

Definition 2.2.3 (Regular Algebra) *A boolean algebra with an additional product operation which distributes over union is called a regular algebra.*

2.3 The quotient of languages.

The possible values of variables whose values are representations of sequences are characterised by languages. We often manipulate such sequences by taking things off one end or the other. For example, an operation that takes things off a stack will make any sequence of objects which constitute the state of the stack *shorter*. We can characterise removal from the right by the *right quotient* operation.

Definition 2.3.1 *If L and K are languages then the right-quotient L/K is defined by (H&U p62):*

$$L/K = \{x | \exists y \in K, xy \in L\}$$

There is, symmetrically, a left quotient operation, which we shall discuss briefly later.

Lemma 2.3.1 (Monotonicity of Division) *Let $L_1 \subseteq L_2$ and L be languages. Then*

$$L_1/L \subseteq L_2/L$$
$$L/L_1 \subseteq L/L_2$$

Proof: Let $x \in L_1/L$ Then there exists $l \in L$ for which $xl \in L_1$, Hence $xl \in L_2$, that is $x \in L_2/L$. Hence $L_1/L \subseteq L_2/L$.

Let $x \in L/L_1$. Then there exists $l_1 \in L_1$ for which $xl_1 \in L$ But $l_1 \in L_2$. Hence $x \in L/L_2$.

While the first monotonicity relation above is certainly to be expected, the second will be surprising to many readers.

We use the convention that multiplication is more binding than division, so that $LK/L'K'$ means $(LK)/(L'K')$.

Being able to perform division of languages is an essential requirement for our type-checker. If L and K are *regular* languages, then the quotient is *regular*. However, we need to be able to divide languages which contain variables. Typically this will be possible in circumstances in which some kind of special *marker* has been used which is known not to occur in the language denoted by the variable. In POP-11, this arises in the analysis of functions like `sysconslist`, whose type-signature is:

$$\mathtt{sysconslist : All\ a;\ Stackmark * a\ -> \ List(a)}$$

Here $\mathtt{a}$ is a variable which denotes a language none of whose tokens are the same as those of the monolog `Stackmark`.

The strategy of the type-checker in performing division is to express division of complex regular expressions in terms of divisions of simpler expressions, that is to say, the division operation is moved *inwards*. If necessary, divisions which cannot currently be performed are deferred. Such deferral is required in inferring the type of recursive functions.

Division has only weak algebraic properties. In particular, cancellation does not hold in general. That is LK/K is *not* in general equal to L. This arises from the fact that if $l \in L$ and $ly \in LK$, we *cannot* conclude that $y \in K$, since y might have the form gk, where $lg \in L$, $k \in K$. However we can readily see that if K is a *monolog*, then cancellation must be possible, an instance of proposition2.3.1 below.

Indeed, it is not even true that $L/L = \mathbf{1}$. Suppose $L = \{a\}$. $L/L = \{x|\exists y \in L, xy \in L\} = L$. The essential strategy in the division of products is to find conditions under which cancellation does occur.

Definition 2.3.2 (Exact Division) *We say that J is* exactly divisible *by K if every $j \in J$ has the form gk where $k \in K$.*

When a procedure takes its arguments off the stack, for type-correctness it is essential that the argument language exactly divides the stack-language.

Being able to divide one language by another is crucial to our type-system. It is easy to see that a simple divide-and-conquer strategy cannot be applied because of the fact that cancellation does not always work for products. The lemmas below suggest that divide-and-conquer has some hope of being successful for sums. In dealing with products, the general strategy will be to consider *monologs*, as discussed below.

Lemma 2.3.2 *If J, K, L are languages, then $(J \cup K)/L = J/L \cup K/L$, and the left-hand division is exact if and only if both of the right hand divisions are exact.*

Proof: Let $x \in (J \cup K)/L$ Then there is $l \in L$ for which $xl \in J \cup K$ Suppose $xl \in J$ then $x \in J/L$. Alternatively, if $xl \in K$ then $x \in K/L$. In either case $x \in J/L \cup K/L$,

Conversely, suppose $x \in J/L \cup K/L$. Suppose $x \in J/L$. Then there is an $l \in L$ for which $xl \in J$. Thus $xl \in J \cup K$. So $x \in (J \cup K)/L$. Alternatively, if $x \in K/L$ we conclude similarly that $x \in (J \cup K)/L$.

Now suppose $J \cup K$ is exactly divisible by L. Let $x \in J$, then $x \in J \cup K$ so that $x = x'l$ where $l \in L$. Thus J is exactly divisible by L. Likewise K is exactly divisible by L.

Conversely, if both J and K are exactly divisible by L, let $x \in J \cup K$ Then if $x \in J$ it follows that $x = x'l$ for some $l \in L$. Likewise if $x \in K$, $x = x'l$ for some $l \in L$. Therefore $J \cup K$ is exactly divisible by L.

Lemma 2.3.3 *If J, K, L are languages, then $L/(J \cup K) = L/J \cup L/K$, and the left-hand division is exact if one of the right hand divisions is exact.*

Proof: Let $x \in L/(J \cup K)$. Then, there is $y \in J \cup K$ for which $xy \in L$. Suppose $y \in J$. Then $x \in L/J$. Alternatively, if $y \in K$ then $x \in L/K$. Thus $x \in L/J \cup L/K$

Conversely, let $x \in L/J \cup L/K$. Suppose $x \in L/J$. Then there is a $j \in J$ for which $xj \in L$. But $j \in J \cup K$. So $x \in L/(J \cup K)$. Similarly we see that if $x \in L/K$ then $x \in L/(J \cup K)$.

Now suppose J exactly divides L. Then if $l \in L$, there is a $j \in J$ and an $l' \in L$ for which $l = l'j$. But $j \in J \cup K$ Hence $J \cup K$ exactly divides L. Likewise, if K exactly divides L, $J \cup K$ exactly divides L.

Note that in distinction to the previous lemma, we cannot conlude the converse fact about exact division.

Sometimes, as we shall see, it is possible to perform division easily and explicitly by relying on the fact that, in our application, many languages are known to be monologs.

Lemma 2.3.4 *If J, K, L are languages then*

$$J/KL = (J/L)/K$$

Moreover, if L is a monolog, then KL exactly divides J if and only if L exactly divides J and K exactly divides J/L

Proof: Let $x \in J/KL$. Then there is $kl \in KL$ for which $xkl \in J$. Hence $xk \in J/L$, and so $x \in (J/L)/K$.

Conversely, if $x \in (J/L)/K$ then there is a k for which $xk \in J/L$, that is there is a l for which $xkl \in J$, that is $x \in J/KL$.

Now suppose L is a monolog, and KL exactly divides J. Consider $j \in J$. Then, for some $x, k \in K, l \in L$, $j = xkl$ Thus L exactly divides J.

Consider now $x \in J/L$. Then there is an $l \in L$ for which $xl = j \in J$. Hence, since KL exactly divides J, $xl = x'k'l'$, for some x', $k' \in K$, $l' \in L$. But L is a monolog, so l and l' have length 1. Hence $x = x'k'$ and hence K exactly divides J/L.

Finally, suppose L exactly divides J and K exactly divides J/L. Consider j in J. Then $j = x'l$, for some x', $l \in L$. Now $x' \in J/L$ so $x' = x''k$, for some x'', $k \in K$. Thus $j = x''kl$, so that KL exactly divides J.

Proposition 2.3.1 *Let J, J' be languages. Let $M \subseteq N$ be monologs, and let $K \subseteq L$ be non-empty languages for which $\Sigma_0(N) \cap \Sigma_0(L) = \emptyset$. Then $JMK/J'NL = J/J'$ and the division is exact iff J' exactly divides J.*

Proof: Let $x \in JMK/J'NL$. Then there is a $j'nl \in J'NL$ for which $xj'nl = jmk \in JMK$. But, m is not a member either of the sequences k, l and neither is n. Hence $k = l$, and $xj' = j$. So $x \in J/J'$.

Conversely, suppose $x \in J/J'$. So there is a $j' \in J'$ for which $xj' \in J$. Then, since M is a monolog, and K is non-empty, there exist $m \in M$, $k \in K$. Hence $xj'mk \in JMK$. Remembering that $M \subseteq N$ and $K \subseteq L$ we see that $j'mk \in JNL$, that is $x \in JMK/J'NL$. Thus $JMK/J'NL = J/J'$.

Now suppose $J'NL$ exactly divides JMK. Consider $j \in J$. Now, with m, k from the previous paragraph $jmk \in JMK$. So there exist x, $j' \in J', n \in N, l \in L$ for which $jmk = xj'nl$. But m is not a member of the sequences l, k. Neither is m'. Hence $j = xj'$, showing that J' exactly divides J.

Conversely, suppose J' exactly divides J. Consider $jmk \in JMK$. By the definition of exact division, $j = xj'$ for some $j' \in J'$. So $xj'mk \in JMK$, and, because of our inclusion relation, $j'mk \in J'NL$. Hence $J'NL$ exactly divides JMK.

Corollary 2.3.1 *Let J, J' be languages, for which J' exactly divides J and let $M \subseteq N$ be monologs. Then $J'N$ exactly divides JM and*

$$JM/J'N = J/J'$$

For we can take $K = L = \mathbf{1}$ in the previous proposition.

Corollary 2.3.2 *Let L be a language, $M \subseteq N$ be monologs. Then N exactly divides LM and*

$$LM/N = L$$

For we can take $J = \mathbf{1}$, $K = L$ in the above corollary.

Corollary 2.3.3 *Let $M \subseteq N$ be monologs. Then N exactly divides M and*

$$M/N = \mathbf{1}$$

For we can take $J = K = \mathbf{1}$ in the above corollary.

Proposition 2.3.2 *Let J, J' be non-empty languages. Let M, N be monologs, and let K, L be non-empty languages for which $\Sigma_0(M) \cap \Sigma_0(K) = \emptyset$. $\Sigma_0(M) \cap \Sigma_0(L) = \emptyset$. $\Sigma_0(N) \cap \Sigma_0(K) = \emptyset$. $\Sigma_0(N) \cap \Sigma_0(L) = \emptyset$. Let $J'NL$ exactly divide JMK. Then $K \subseteq L$ and $M \subseteq N$.*

Proof: Let $k \in K$. Since J, M are non-empty, let $j \in J, m \in M$, so $jmk \in JMK$, Then, by the definition of exact division, there is an x for which $jmk = xj'nl$, $j' \in J', n \in Nl \in L$. Hence, as before, $l = k$, that is $k \in L$, showing that $K \subseteq L$.

Now let $m \in M$. Since J, K are non-empty, let $j \in J, k \in K$, so $jmk \in JMK$, Then, by the definition of exact division, there is an x for which $jmk = xj'nl$, $j' \in J', n \in Nl \in L$. Hence, $m = n$, that is $M \subseteq N$, showing that $K \subseteq L$.

2.4 The Kleene Closure in Division

The Kleene Closure operation presents the most difficulties for division. ¿From proposition 2.3.3 we see, for example that

$$K/\{L\}_n = \bigcup_{i=n}^{\infty} K/L^i$$

It is known that when both K and L are regular languages, there is a computable i beyond which no new terms are generated in the union, so that we could thus get rid of Kleene Closures in the 'denominator'. However, since we are developing our theory with no assumption of regularity, we are left with weaker conclusions.

In the case when we are dividing a monolog by a Kleene Closure, we have the following:

Proposition 2.4.1 *Let M be a monolog, L, K be languages. Then*

$$LM/\{K\} = L/(\{K\}(K/M)) \cup LM$$

Proof: Let $x \in LM/\{K\}$. Then there is $k_1 \ldots k_n \in K$ for which $xk_1 \ldots k_n = l\hat{m} \in LM$.

Supposing $n > 0$, then

- *either* there is $k_{n'}$ which is the rightmost $k_i \neq \epsilon$ $k_{n'} = \hat{y}_1 \ldots \hat{y}_j$ Then $\hat{y}_j = \hat{m}$ so that $\hat{y}_1 \ldots \hat{y}_{(j-1)} \in K/M$, and $xk_1 \ldots k_{n-1}\hat{y}_1 \ldots \hat{y}_{j-1} = l$ for some $l \in L$, Hence $x \in L/(\{K\}(K/M))$
- *or* $k_i = \epsilon$ for all $i \in 1 \ldots n$. Here $x = l\hat{m} \in LM$.

  If $n = 0$, then $x = l\hat{m} \in LM$.

  Conversely, let $x \in L/(\{K\}(K/M)) \cup LM$

  Suppose $x \in L/(\{K\}(K/M))$. Then, for some $k_1 \ldots k_n \in K$ and $y \in K/M$ $xk_1 \ldots k_n y \in L$. And, for some $\hat{m} \in M$, $y\hat{m} \in K$. So $xk_1 \ldots k_n y\hat{m} \in LM$. Hence $x \in LM/\{K\}$.

  If, on the other hand $x \in LM$, then, since the empty sequence $\epsilon \in \{K\}$, $x \in LM/\{K\}$

  We next address the problem of a Kleene closure in the 'numerator'. Note that the following proposition is stronger than

$$L\{K\}/M = L/M \cup (L\{K\}K)/M$$

  which can be inferred by expanding out the Kleene closure using 2.2.1

Proposition 2.4.2 *Let M be a monolog, L, K be languages, then*

$$L\{K\}/M = L/M \cup L\{K\}(K/M)$$

Proof: Let $x \in L\{K\}/M$. Then, for some $\hat{m} \in M$, $l \in L$ $k_1 \ldots k_n \in K$, $x\hat{m} = lk_1 \ldots k_n$. Now, if $n = 0$, or $k_i = \epsilon$ for $i \in 1 \ldots n$ then $x\hat{m} \in L$, so $x \in L/M$. Otherwise, let $k'_n = \hat{y}_1 \ldots \hat{y}_j$ be the rightmost non-null sequence of the k_i. Thus $\hat{y}_j = \hat{m}$, and $\hat{y}_1 \ldots \hat{y}_{j-1} \in K/M$. Hence $x \in L\{K\}(K/M)$

Conversely, suppose $x \in L/M \cup L\{K\}(K/M)$. If $x \in L/M$ then $x \in L\{K\}/M$, since the Kleene closure contains the empty seqence. Alternatively, $lk_1 \ldots k_n x \in L\{K\}(K/M)$ where $x\hat{m} \in K$, for some $\hat{m}$, so that $lk_1 \ldots k_n x\hat{m} \in L\{K\}$. Thus $lk_1 \ldots k_n x \in L\{K\}/M$.

2.5 Converting divisions into inequalities

In circumstances where no cancellation is possible the following inequalities are of use:

Lemma 2.5.1 *If K is non-null, then $L \subseteq (LK)/K$*

Proof: Let $l \in L$. Let $k \in K$. $lk \in LK$ so that $l \in (LK)/K$

Lemma 2.5.2 *If J is exactly divisible by K then $J \subseteq (J/K)K$*

Proof: Let $j \in J$. Then, $j = gk$, so $g \in J/K$ by the definition of division. Hence $j \in (J/K)K$.

2.6 Left Division - the *Rev* function

While we have regarded *stacks* as being operated on *from the right* and so have developed *right division*, there are structures, such as lists, that we operate on *from the left* with the traditional *hd* and *tl* operations. In POP-11 this is not just a matter of convention, since there are constructs like the `[% .... %]` form which convert between a right-access structure (the stack) and a left-access structure (a list). Thus we are going to need a concept of *left division*.

Since there is no intrinsic 'direction' to the sequences of our languages, we could develop the concept of *left division* in a manner exactly analogous to our treatment of *right division*. However we prefer the treatment below.

Definition 2.6.1 *The language-function Rev is defined by $x \in Rev(L)$ if and only if $rev(x) \in L$, where $rev(a_1 \ldots a_n) = (a_n \ldots a_1)$.*

It is clear that *Rev* is monotonic.

Definition 2.6.2 (left-quotient)

$$K \backslash L = Rev(Rev(L)/Rev(K))$$

Definition 2.6.3

$$Tl(L) = \top_M \backslash L$$

Definition 2.6.4 *If L is a language, then* $\mathrm{Hd}(L) = \{\hat{l} | \exists l'.\ \hat{l}l' \in L\}$

3 The Virtual Machine

The Virtual Machine presented here is related to the Poplog Virtual Machine which has been made the target of a number of compilers. It provides fully automatic (i.e. garbage-collected) storage control, and a range of user functions including arbitrary precision arithmetic, data-structure construction and access, and incremental code generation.

The simplified version of the Poplog Virtual Machine which is defined below is in some ways closer to the original POP-2 machine, and so is close to the 4130 computer. For the type-theory presented here, severe restrictions on that are required. However *lexical locals* characteristic of the PVM are required, as opposed to the dynamic locals of POP-2 [3]. We characterise the VM in terms of a function *exec* which executes one instruction, *exec** which executes instructions repeatedly, and *obey* which obeys a block of code.

If θ is a mapping, then we use $\phi = \theta[x \mapsto y]$ to mean a mapping for which $\phi(x) = y$ and $\phi(x') = \theta(x')$ otherwise. We use $dom(\theta)$ to mean the *domain* of θ. The function $\theta[x \mapsto y]$ may, or may not, have a domain bigger than that of θ.

A machine state consists of a quintuple $(s, \mathbf{e}, c, b, h)$.

Here, h is a *heap*, that is to say, a mapping from *addresses* to values. Addresses are drawn from a countably infinite set A, but the domain of any given heap is *finite*. A value is either (a) a pair $(a_{ProcKey}, p)$, where p is a primitive function or (b) a tuple $(a_0, \ldots a_n)$. Here the first element of a tuple is always the address of a *key value*, which determines the *basic type* of the object. A tuple can be a code-block, in which case the components $a_1 \ldots a_n$ are *instructions* as described below. Otherwise they are addresses.

There is always an undefined object `u`, and true and false objects `t` and `f` in any heap.

s is the stack, that is to say a tuple of addresses.

$W \subseteq dom(h)$ is a set of *identifiers*. No assumption is made here about whether $W \subset A$, as in POP-2, or not, as is the case in most other languages, where identifiers are not run-time objects. However, we do not treat those language-capabilities that hinge on identifiers being run-time objects, namely the `valof` and `popval` functions.

$\mathbf{e}$ is an environment, that is to say a sequence of mappings from identifiers to addresses. An identifier is either *local* to a code-block or global. Non-locals which are not global are eliminated by lambda-lifting.[1]

Definition 3.0.5 (Value of identifier in environment) *The value* $\mathbf{e} \otimes w$ *of a identifier* w *is* $\mathbf{e}_0(w)$ *if* w *is non-local to a code-block and* $\mathbf{e}_n(w)$ *if* w *is local, where* $\mathbf{e}_n$ *is the last member of the sequence. A variable is bound to* `u` *to indicate that it is 'undefined'.*

[1]The POP-2 manual instructed programmers to eliminate problematic free variables by making them arguments of the function in which they occurred, which was then to be partially applied to give a closure. This approach could have avoided the need for globals entirely, but while globals are not logically necessary, they are important for efficiency.

We use $\mathbf{e}[v \mapsto y] = \mathbf{e}[i \mapsto \mathbf{e}_i[v \mapsto y]]$ *where* $i = 0$ *for a non-local variable and* $i = n$ *for a local variable.*

b is a code-block, that is to say a sequence of instructions. c is an index into b, which determines the current instruction. Associated with each code-block are its lexical variables. The function *lvars* makes a new environment from $\mathbf{e} = e_1 \ldots e_n$ and a code-block. $lvars(\mathbf{e}, b) = (e_0 \ldots e_{n+1})$ where $e_{n+1} = [v_1 \mapsto \mathtt{u}, \ldots v_k \mapsto \mathtt{u}]$ for all the local variables of b.

There is a set of *labels* which are distinct from the set of identifiers.

In the following two sub-sections we will define the operation of the virtual machine. We first define a function *exec* which says what it means to execute a single instruction of the VM. We then define *obey* which says what it means to obey a code block of the virtual machine.

3.1 Executing an Instruction

The instruction set of the VM is: $pushq(n)$, where $n \in h$, $push(w)$, $pop(w)$ where $w \in W$. $call(w)$, $label(l)$, $goto(l)$, $ifnot(l)$ where l is a *label*. The instruction with a given label is unique within a code-block.

It is always possible that a *call* may not terminate. To deal with this case we introduce $\perp$ to be the value of a non-terminating computation. The execution of an instruction is achieved by the *exec* function, defined below. $exec_d(s, e, c, b, h)$ means 'execute the instruction at location c of code-block b, with stack s, environment e and heap h'. $d \geq 0$ is the *depth-bound* of the computation — if there are $\geq d$ nested calls, then *obey* will evaluate to $\perp$.

$$\begin{array}{lcll}
exec_d(s, \mathbf{e}, c, b, h) & = & (s\hat{n}, \mathbf{e}, c+1, b, h) & \text{if } b(c) = pushq(n) \\
exec_d(s, \mathbf{e}, c, b, h) & = & (s\mathbf{e} \otimes w, \mathbf{e}, c+1, b, h) & \text{if } b(c) = push(w) \\
exec_d(s'\hat{n}, \mathbf{e}, c, b, h) & = & (s', \mathbf{e}[w \mapsto n], c+1, b, h) & \text{if } b(c) = pop(w) \\
exec_d(\epsilon, \mathbf{e}, c, b, h) & = & \perp & \\
exec_d(s, \mathbf{e}, c, b, h) & = & (s', \mathbf{e}', c+1, b, h') & \text{if } b(c) = call(w) \\
 & & \multicolumn{2}{l}{\text{where } (s', \mathbf{e}', h', \mathbf{S}) = obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d-1)} \\
exec_d(s, \mathbf{e}, c, b, h) & = & \perp & \text{if } b(c) = call(w) \text{ and} \\
 & & \multicolumn{2}{c}{obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d-1) = \perp} \\
exec_d(s, \mathbf{e}, c, b, h) & = & (s, \mathbf{e}, c+1, b, h) & \text{if } b(c) = label(l) \\
exec_d(s, \mathbf{e}, c, b, h) & = & (s, \mathbf{e}, c'+1, b, h) & \text{if } b(c) = goto(l) \\
 & & \text{where } b(c') = label(l) & \\
exec_d(s\hat{\mathtt{f}}, \mathbf{e}, c, b, h) & = & (s, \mathbf{e}, c'+1, b, h) & \text{if } b(c) = ifnot(l) \\
 & & \text{where } b(c') = label(l) & \\
exec_d(s\hat{n}, \mathbf{e}, c, b, h) & = & (s, \mathbf{e}, c+1, b, h) & \text{if } b(c) = ifnot(l) \\
 & & \text{and } \hat{n} \neq \mathtt{f} & \\
exec_d(\epsilon, \mathbf{e}, c, b, h) & = & \perp & \text{if } b(c) = ifnot(l)
\end{array}$$

3.2 Obeying a whole code block or primitive

The definition of *obey* depends on whether a primitive or a code-block is being obeyed. If p is a primitive, then it acts on the stack and the heap to produce a new stack and heap.

$$obey(s, \mathbf{e}, (a_{ProcKey}, p), h, d) = (s', \mathbf{e}, h', \epsilon) \text{ where } p(s, h) = (s', h') \tag{1}$$

Note that a *new* heap is created as the result of obeying a primitive. It is clear that this new heap must resemble the old one strongly if we are to be able to make any predictions whatsoever about the behaviour of the VM.

¿From the point of view of type-checking, the important thing is that a primitive cannot change the type of any existing object. Since we are allowing parametric polymorphism this is a tricky condition to maintain if objects are mutable. This paper deals only with the case of immutable objects. That is to say, in the above definition $h' = h[a_1 \mapsto (a_{10} \ldots a_{1n_1}) \ldots a_m \mapsto (a_{m0} \ldots a_{mn_m})]$ where $a_1 \ldots a_m \notin dom(h)$.

If b is a code-block, then

$$obey(s, \mathbf{e}, b, h, 0) = \bot \tag{2}$$

$$obey(s, \mathbf{e}, b, h, d, \mathbf{S}) = exec^*_d(s, lvars(\mathbf{e}, b), b, h) \tag{3}$$

Here the $exec^*_d$ is defined as follows. Suppose there is a sequence of states $\mathbf{S} = S_0 \ldots S_n$ where $S_0 = (s, \mathbf{e}, 1, b, h)$ and $S_{i+1} = exec_d(S_i)$ and $S_n = (s', \mathbf{e}', c, b, h')$ where $c = |b| + 1$ (that is we have 'run off the end of the code block'). Then $exec^*_d(S_0) = (s', \mathbf{e}', h', \mathbf{S})$.

The following two lemmas are trivial consequences of the definition of the VM.

Lemma 3.2.1 *If* $obey(s, \mathbf{e}, b, h, d) = (s', \mathbf{e}', h', \mathbf{S})$ *then*

$$obey(s''s, \mathbf{e}, b, h, d) = (s''s', \mathbf{e}', h', \mathbf{S}')$$

where $\mathbf{S}'$ *is a state-sequence.*

Lemma 3.2.2 *If* $\mathbf{e}$ *and* $\mathbf{e}'$ *are environments where* $\mathbf{e}_0 = \mathbf{e}'_0$. *Let* $obey(s, \mathbf{e}, b, h, d) = (s', \mathbf{e}'', h', \mathbf{S})$. *Then*

$$obey(s, \mathbf{e}', b, h, d) = (s', \mathbf{e}''', h', \mathbf{S}')$$

where env''' *is an environment for which* $\mathbf{e}''_0 = \mathbf{e}'''_0$

Definition 3.2.1 (Closed VM State) *We say that a VM state S is closed if for every instruction of the form $call(w)$ occurring in any code-block b of the heap of S, either w is a local variable of b, or $h(\mathbf{e} \otimes b) = (a_{ProcKey}, \ldots)$ is a code-block.*

3.3 Record and Vector Keys

Objects on the heap are records, of fixed size, vectors of variable size and code-blocks. At the level of abstraction of this paper, all we need to know about a record-key is that it has an associated *arity*, which determines the number of components of each record of the class determined by the key.

4 Descriptions of the VM

The purpose of our formal characterisation of the VM is to allow us to make predictions about its behaviour. Any prediction about the behavour of the VM requires us to to be able to *describe* machine states and *annotate* code blocks. This is true whether whatever the nature of the predictions we wish to make — they may serve for type-checking as in the present paper or for less limited kinds of program verification.

4.1 Discussion of annotation

The issue of annotation is quite a complex one. Essentially, we want to annotate a code-block with an adequate description of the state of the machine at each instruction. Here, "adequate" means adequate from the point of view of type-checking, that is to say determining whether operations are legal.

The descriptive apparatus *must* include a description of the stack, and an environment which describes the types of VM variables. The primary problem is that we need type-variables. These will occur both in the stack-description and the environment; certain type-variables will be known to be monolog-valued. And constraints on variables will be generated in various ways. Consider the POP-11 sequence `...hd(L) + 2 -> L1;...`, which translates into the code-block

$$b = (\dots push(L), call(hd), pushq(2), call(+), pop(L1), \dots)$$

Now, at $b_1 = push(L)$ we may know nothing about the type of L, that is to say its type is a monolog-valued variable v. Assuming the stack is empty on entrance to the sequence, as we start to execute b_2, the stack is described by v. At b_2 we try to divide v by the argument-type of `hd`, which is $List(v_1)$, where v_1 is a polylog-valued type-variable. Since *List* is a monolog-valued type-function, we infer that $v \subseteq List(v_1)$. We can express this most generally by *unifying* v and $List(v_1)$, with the substitution $v \mapsto List(v_1)$, since anything additional we might infer about v can be expressed in terms of a constraint on v_1. For example, when we come to $call(+)$, we can infer that $\mathrm{Hd}(v_1) \subseteq$ `Number`.

The procedure for deriving an annotation must thus examine the instructions of a code-block, inferring constraints on type-expressions. The order of examination should be related to that of execution. When all information relevant to a type-variable has been garnered into constraints, an attempt will be made to resolve them, deriving a value for the variable. Within this framework we can treat jumps and labels. At a *label* instruction, the stack at the label will derived from the stack at any instruction from which the label can be reached — this can be treated by introducing a variable for the stack at the label and a *production* for each way of getting to the label. These productions are also, in effect, inequality constraints.

Another issue that must be addressed is the necessity of deriving more restricted types for some variables arising from the need to exploit the fact that some POP-11 functions serve to *recognise* data-structures. Consider for example the sequence `if isnumber(x) then x+3 else...` We may conclude that `x` is a number between the `then` and `else`. To deal with this issue, we introduce the concept of *recognisers* into the type-system, introduce a restricted *local typing*, and also, in a strictly limited way maintain an expression (or term) which denotes the actual value of the top-of-stack.

4.2 Grammars characterise languages

So far we have treated languages abstractly as possibly infinite sets of sequences. In order to compute with such entities, we have to find a finite representation. This is provided by *grammars*.

Languages can be represented most generally by unrestricted grammars. But we want to keep things computable, so aim at regular grammars. How unrestricted might our grammars be? First let us observe that a production, $\alpha \rightarrow \beta$ can be construed as an inequality $\beta \subseteq \alpha$. Thus the grammatical quotient, $L = J/K$, which, if exact, gives rise to the inequality $J \subseteq LK$ can be seen as having the corresponding production $LK \rightarrow J$. Thus we have have unrestricted grammars lurking in the undergrowth.

However, we are aiming at reducing all our grammars to being regular algebras. This means that we will introduce productions as a necessary part of the process of type-checking a VM program. But they will be eliminated as a requirement that the program is type-correct.

In this section we

1. Introduce *extended regular word algebras* as our formalism for expressing type.
2. Introduce *type-environments* as a way of recording information about type.
3. Define the terminal symbols of our type-formalism as characterising objects of the VM.
4. Define a function $\mathcal{L}$ which maps a *type-expression* into a *type-language* of sequences drawn from a *heap* and using an *environment*.

Definition 4.2.1 *An extended regular word algebra (ERWA) is a set of expressions formed from a set of monolog-variables V_m, a set of polylog-variables V_p and a set of terminals T.*

$\mathcal{R}(V_m, V_p, T)$ *consists of expressions formed as follows.*

- $\emptyset \in R$
- $\mathbf{1} \in R$
- If $v \in V_m \cup V_p$ then $v \in R$.
- If $t \in T$ then $t \in R$
- If $r_1, r_2 \in R$ then $r_1 \rightarrow r_2 \in R$
- If $r_1, r_2 \in R$ then $r_1 \cup r_2 \in R$
- If $r_1, r_2 \in R$ then $r_1 r_2 \in R$
- If $r_1, r_2 \in R$ then $r_1 / r_2 \in R$
- If $r_1 \in R$ then $\{r_1\} \in R$

If $R = \mathcal{R}(V_m, V_p, T)$ is an ERWA, then we can associate a language with R by saying what the variables and terminals "mean". Just as we needed an environment to say what values variables have in the VM, so we need a kind of environment to tell us things about types. This includes both *type equations and productions* which will tell us about what type variables mean, but also, to analyse the type of code-blocks, we will need to know about the types of variables in the VM itself, and also whether certain variables have values which can be used to *recognise* objects.

Definition 4.2.2 *Let R be an ERWA. We say that an expression $r \in R$ is quotient-free if no sub-expression of r is of the form r_1/r_2*

The expressions for a legal type will be quotient-free.

Definition 4.2.3 (Type Environments) *A type-environment* $\mathbf{E}$ *is a quadruple*

$$(\mu, \nu, \phi, P)$$

where $\mu : W \rightarrow R$ specifies the type of variables. $\nu : W \rightarrow R$ specifies that a variable recognises a type, $\phi : V \rightarrow R$ specifies type-equations and P is a set of productions *of the form $v \rightarrow r$ where $v \in V_p$ and $r \in R$.*

In the above definition, μ is also written $\mathbf{E}_{:}$, ν is also written $\mathbf{E}_{?}$, ϕ is also written $\mathbf{E}_{=}$ and P is also written $\mathbf{E}_{\rightarrow}$.

The terminals of our type-language are monologs, the members of which are drawn from a set of objects of the VM, that is to say a set of addresses. There are two kinds of terminal, namely singletons and data-classes. A *singleton* consists of just one sequence consisting of a given address. A data-class is drawn from a set of objects which have the same key and may have additional commonality in their components.

Definition 4.2.4 (singleton) *A singleton-expression for an ERWA R is a term $S(a)$, where a is an address.*

Definition 4.2.5 (dataclass) *A dataclass expression is a term $D(a, r_1 \dots r_n)$ where a is an address, and $r_1 \dots r_n \in R$*

We are now ready to define how an expression of an ERWA denotes a language. The language denoted by an expression r *is* a set of sequences of addresses. Therefore we need hardly be surprised that, as well as a type-environment to tell us what the variables mean, we will also need a heap to allow us to interpret the addresses.

However we need to define a meaning for the form $r_1 \to r_2$. This is usually thought of as a function type, and will in our case denote a monolog whose members are *code-blocks*, since these fill the role of functions. This does imply a considerable complication of our system. In order to make any predictions about type we will need significant *type stability* of objects with respect to the *exec* and *obey* functions, and also with respect to the compiler operating as *deus ex machina*. That is to say, for us to be able to reason about the types of variables, an object to which a variable is bound must retain its type throughout the period for which the variable is bound to it. The more precise and specific the characterisation of type, the less we will be able to do to an object without changing its type as characterised.

The type of a code block must depend on the way it relates its arguments to its results. So we must provide at least one machine-state for us to observe its execution. However the case is worse than this, for the behaviour of a code block can change significantly with small changes in the state. Consider the POP-11 procedure:

```
define fred(x);
  if x>2 then 'big' else x
  endif
enddefine;
```

If a heap h contains only the integers `0` and `1` say, then we may observe that `fred`, when given an integer argument returns an integer result in any state which has h as its heap. That is to say, observationally `fred:Int->Int`. However if we form $h' = h[a \mapsto 3]$, for some address $a \notin dom(h)$ then `fred` may return a string when given an integer. This is such a trivial change in h that we would be completely unable to construct a type-theory.

Thus it appears that we should not attempt to assign a type to a code-block by observing its behaviour with respect to just one machine-state. It is equally problematic if we attempt to make use of all machine-states, since the behaviour of a code-block depends upon its non-local variables, and if we range over all machine-states, we could get almost any behaviour out of a code-block. Happily there is a middle way. We can define the type of a code-block in terms of its behaviour in a given state, and all states which can be *reached* from that state via a sequence of *legal transitions*. What constitutes a legal transition can be decided later. Indeed different type-theories will be associated with different kinds of legal transition. The more restrictive we make the definition of a legal transition, the stronger our type-theory will be.

Definition 4.2.6 (Legal Reachability) *A transitive relation $\Longrightarrow$ defined over the set of VM states is said to be a legal-reachability relation if*

1. *$\Longrightarrow$ is transitive*
2. *If S is a VM state, then $S \Longrightarrow exec_d(S)$.*
3. *If $S = (s, \mathbf{e}, c, b, h)$ is a VM state, and $S' = (s, \mathbf{e}, c, b, h[a' \mapsto (a_{key}, a_1, \ldots a_n)])$, where $a' \notin dom(h)$ then $S \Longrightarrow S'$.*

Definition 4.2.7 (Non-mutating reachability) *A reachability relation $\Longrightarrow$ is said to be non-mutating if*

$$S = (s, \mathbf{e}, c, b, h) \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$$

implies that if $a \in dom(h)$ then $a \in dom(h')$ and $h'(a) = h(a)$

Definition 4.2.8 (Manifest monolog) *We say that $r \in R$ is a manifest monolog if it is of the form $S(a)$, $D(a_{key} \ldots)$ or v where $v \in V_m$.*

We are now in a position to define the denotation of type-expressions.

Definition 4.2.9 ($\mathcal{L}$ maps from expressions to languages) *If $r \in R$, $\mathbf{E}$ is a type-environment and $S = (s, \mathbf{e}, c, b, h)$ is a machine-state, and $\Longrightarrow$ is a legal-reachability relation then $\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ is defined for integer depth $d \geq 0$ as follows:*

1. $\mathcal{L}(\mathbf{E}, \mathbf{1}, S, \Longrightarrow, d) = \mathbf{1}$
2. $\mathcal{L}(\mathbf{E}, \emptyset, S, \Longrightarrow, d) = \emptyset$
3. For any object $a \in dom(h)$, $\mathcal{L}(\mathbf{E}, S(a), S, \Longrightarrow, d) = \{\hat{a}\}$
4. For any key-object $a_{key} \in dom(h)$, if a_{key} is a record-key of arity n, then

$$\mathcal{L}(\mathbf{E}, D(a_{key}), S, \Longrightarrow, d)$$

$$= \{\hat{a} | h(a) = (a_{key}, a_1 \ldots a_n) \text{ where } a_1 \ldots a_n \in dom(h)\}$$

.

5. For any key-object $a_{key} \in dom(h)$, if a_{key} is a record-key of arity n , and $r_1 \ldots r_n \in R$ are type-expressions for which $L_i = \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d)$ are manifest monologs, then

$$\mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S, \Longrightarrow, d) = \{\hat{a} | h(a) = (a_{key}, a_1 \ldots a_n), \hat{a}_i \in L_i)\}$$

6. For any vector-key-object $a_{key} \in dom(h)$, and $r \in R$, then

$$\mathcal{L}(\mathbf{E}, D(a_{key}, r), S, \Longrightarrow, d) =$$

$$\{\hat{a} | h(a) = (a_{key}, a_1, \ldots a_n), (a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d)\}$$

7. Otherwise $\mathcal{L}(\mathbf{E}, D(a, r_1 \ldots r_n), S) = \emptyset$
8. For any $r_1, r_2 \in R$,

$$\mathcal{L}(\mathbf{E}, r_1 \to r_2, S, \Longrightarrow, d) = L \tag{4}$$

   where $L =$
   $\{\hat{a} | h(a) = (a_{ProcKey}, \ldots),$
   $\forall S' \, \forall d_1 \leq d.$ if $S \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$ then
   $l_1 \in \mathcal{L}(\mathbf{E}, r_1, S', \Longrightarrow, d_1)$ implies $l_2 \in \mathcal{L}(\mathbf{E}, r_2, S'', \Longrightarrow, d_1)$
   where $(l_2, \mathbf{e}'', h'', \mathbf{S}) = \mathit{obey}(l_1, \mathbf{e}', h'(a), h', d_1)$ and $S'' = (l_2, \mathbf{e}'', c', b', h'')$
   $\}$
9. if $v \in V$ and $\mathbf{E}_{=}(v) = r \neq \perp$ then $\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.
10. Otherwise, if $v \in V$ and $\exists v | v \to r \in \mathbf{E}_{\to}$ then

$$\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \bigcup \{\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) | v \to r \in \mathbf{E}_{\to}\} \tag{5}$$

11. Otherwise, if $v \in V_m$, $\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top_M$.
12. Otherwise, if $v \in V_p$, $\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top$.
13. $\mathcal{L}(\mathbf{E}, r_1 \cup r_2, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r_1, S, \Longrightarrow, d) \cup \mathcal{L}(\mathbf{E}, r_2, S, \Longrightarrow, d)$
14. $\mathcal{L}(\mathbf{E}, r_1 r_2, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r_1, S, \Longrightarrow, d) \mathcal{L}(\mathbf{E}, r_2, S, \Longrightarrow, d)$
15. $\mathcal{L}(\mathbf{E}, r_1 / r_2, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r_1, S, \Longrightarrow, d) / \mathcal{L}(\mathbf{E}, r_2, S, \Longrightarrow, d)$
16. $\mathcal{L}(\mathbf{E}, \{r\}, S, \Longrightarrow, d) = \{\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)\}$

The definition of $\mathcal{L}$ above depends on depth of execution, d. We will be concerned to define the limiting behaviour of $\mathcal{L}$ as d increases. This behaviour is determined by equation 4, where the d parameter is handed to an application of *obey*. We should expect that *as d increases the value of $\mathcal{L}$ decreases*, since *obey* is producing more results which can fail the test of membership of the language denoted by r_2 in equation (4).

Lemma 4.2.1 (Stability of type-environment) *If $\mathbf{E}$ and $\mathbf{E}'$ are type-environments for which $\mathbf{E}_{\to} = \mathbf{E}'_{\to}$ and $r \in R$, S is a machine state, $\Longrightarrow$ is a legal reachability relation and $d \geq 0$ is an integer depth for which $\mathbf{E}_{=}(w) = \mathbf{E}'_{=}(w)$ for all w occurring in either r or $dom(\mathbf{E}_{=})$ or $range(\mathbf{E}_{=})$ or $\mathbf{E}_{\to}$*

$$\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', r, S, \Longrightarrow, d)$$

Proof: This is immediate since the $\mathbf{E}_{:}$ and $\mathbf{E}_{?}$ components of the type environment play no direct or indirect role in the definition of $\mathcal{L}$, and there are no productions or equations which could result in different values.

Lemma 4.2.2 (Stability of local environment) *If* $\mathbf{E}$ *is a type-environment for which* $r \in R$*,* $S = (s, \mathbf{e}, c, b, h)$ *is a machine-state and* $\mathbf{e}'$ *is an environment for which* $\mathbf{e}_0 = \mathbf{e}'_0$*, and* $\Longrightarrow$ *is a non-mutating legal reachability relation and* $d \geq 0$ *is an integer depth then*

$$\mathcal{L}(\mathbf{E}, r, S', \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$$

where $S' = (s, \mathbf{e}', c', b, h)$

Proof: The only way that a change in environment can affect the value of $\mathcal{L}$ is by changing the value of *obey* in equation 4. But, by lemma 3.2.2 this cannot happen.

Definition 4.2.10 (Derivations and their depth) *If* h *is a heap,* $\mathbf{E}$ *a type-environment and* $r \in R$*, then we can* derive $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ *by a number of applications of the rules above. We define the derivation depth* δ *of a derivation by*

- *If* $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ *can be derived by the application of a rule with no reference to* $\mathcal{L}$ *to the right of the equality, then* $\delta = 0$
- *Otherwise* $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ *must be derived by the application of rules which refer to* $\mathcal{L}$ *to the right of the equality. Then* δ *is the one plus the maximum of the derivation depths associated with these references.*

We can now state formally:

Proposition 4.2.1 (Anti-monotonicity of $\mathcal{L}$ with respect to d) *Let* $\mathbf{E}$ *be a type-environment,* $r \in R$ *and let* S *be a machine-state. Let* $\Longrightarrow$ *be a legal-reachability relation. If* $d' \leq d$ *then*

$$\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$$

.

Proof:

We proceed by induction on the derivation-length of $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.

Base case: If the derivation of $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ is of depth 0, then:

- $\mathcal{L}(\mathbf{E}, \emptyset, S, \Longrightarrow, d) = \emptyset$ so l cannot be derived from this.
- If $l \in \mathcal{L}(\mathbf{E}, \mathbf{1}, S, \Longrightarrow, d) = \{\epsilon\}$ then $l \in \mathcal{L}(\mathbf{E}, \epsilon, S, \Longrightarrow, d')$.
- If $l = \hat{a} \in \mathcal{L}(\mathbf{E}, S(a), S, \Longrightarrow, d) = \{\hat{a}\}$ then, $l \in \mathcal{L}(\mathbf{E}, S(a), S, \Longrightarrow, d')$
- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a record-key of arity n. Now

$$\begin{aligned} &\mathcal{L}(\mathbf{E}, D(a_{key}), S, \Longrightarrow, d') \\ &= \{\hat{a'} | h(a') = (a_{key}, a'_1 \ldots a'_n) \mathrm{for some} a'_1 \ldots a'_n \in dom(h)\} \end{aligned}$$

. But $h(a) = (a_{key}, a_1 \ldots a_n)$,
where $a_1, \ldots a_n \in dom(h)$. Hence $l \in \mathcal{L}(\mathbf{E}, D(a_{key}), S, \Longrightarrow, d')$.

Inductive step: Suppose that $\mathbf{E}$ is a type-environment, $r \in R$, and S is a machine-state. Suppose that $d' \leq d$, $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ implies that $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$ provided that the derivation-depth is $\leq \delta$ Now consider a derivation of $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ of depth $\delta + 1$:

- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a record-key of arity n, and $r_1 \ldots r_n \in R$ are type-expressions for which $L_i = \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d)$ are manifest monologs. Now

  $$\mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S, \Longrightarrow, d') = \{\hat{a'} | h(a') = (a_{key}, a'_1 \ldots a'_n), \hat{a'_i} \in L'_i)\}$$

  where $L'_i = \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d')$
  But $\hat{a} \in \{\hat{a'} | h(a') = (a_{key}, a_1 \ldots a_n), \hat{a}_i \in L_i)\}$
  Thus $h(a) = (a_{key}, a_1 \ldots a_n)$, $a_i \in \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d)$.
  But, by the inductive hypothesis, $a_i \in \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d')$;
  hence $\hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S, \Longrightarrow, d')$.

- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a vector-key, and $r \in R$,

  $$\mathcal{L}(\mathbf{E}, D(a_{key}, r), S, \Longrightarrow, d') =$$
  $$\{\hat{a'} | h(a') = (a_{key}, a_1, \ldots a_n), (a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')\}$$

  But $\hat{a} \in \{\hat{a'} | h(a') = (a_{key}, a_1, \ldots a_n), (a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)\}$
  so $h(a) = (a_{key}, a_1, \ldots a_n)$ where $(a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.
  Hence, by the inductive hypothesis, $(a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$.
  Thus we see that $\hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r), S, \Longrightarrow, d')$.

- Consider $\mathcal{L}(\mathbf{E}, D(a', r_1 \ldots r_n), S, \Longrightarrow, d) = \emptyset$, Now $\hat{a} \notin \emptyset$, so that this case is covered by *ex falsa libet*.

- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, r_{arg} \to r_{val}, S, \Longrightarrow, d)$ where $r_{arg}, r_{val} \in R$,
  $\mathcal{L}(\mathbf{E}, r_{arg} \to r_{val}, S, \Longrightarrow, d') =$
  $\{\hat{a'} | h(a') = (a_{ProcKey}, \ldots)$
  and $\forall S' \; \forall d'_1 \leq d'.S \Longrightarrow S', l_{arg} \in \mathcal{L}(\mathbf{E}, r_{arg}, S', \Longrightarrow, d'_1)$
  implies $l_{val} \in \mathcal{L}(\mathbf{E}, r_{val}, S'', \Longrightarrow, d'_1)$
  where $S' = (l_{val}, \mathbf{e}', c', b', h')$
  and $obey(l_{arg}, \mathbf{e}', h'(a'), h', d'_1) = (l_{val}, \mathbf{e}_{val}, h'_{val}, \mathbf{S})$
  and $S'' = (l_{val}, \mathbf{e}_{val}, c', b', h'_{val})\}$
  Now $h(a) = (a_{ProcKey}, \ldots)$ and
  $\forall S' \; \forall d_1 \leq d$ if $S \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$
  then $l_{arg} \in \mathcal{L}(\mathbf{E}, r_{arg}, S', \Longrightarrow, d_1)$ implies $l_{val} \in \mathcal{L}(\mathbf{E}, r_{val}, S'', \Longrightarrow, d_1)$
  where $obey(l_{arg}, \mathbf{e}', h'(a), h', d_1) = (l_{val}, \mathbf{e}'', h'', \mathbf{S})$ and $S'' = (l_{val}, \mathbf{e}'', c', b', h'')$

Since $d' \leq d$, if $d'_1 \leq d'$ then $d'_1 \leq d$ and so for any S' where $S \Longrightarrow S'$, $l_{arg} \in \mathcal{L}(\mathbf{E}, r_{arg}, S', \Longrightarrow, d'_1)$ implies $l_{val} \in \mathcal{L}(\mathbf{E}, r_{val}, S'', \Longrightarrow, d'_1)$
where $obey(l_{arg}, \mathbf{e}', h'(a), h', d'_1) = (l_{val}, \mathbf{e}'', h'', \mathbf{S})$ and $S'' = (l_{val}, \mathbf{e}'', c', b', h'')$
Hence $l = \hat{a} \in \mathcal{L}(\mathbf{E}, r_{arg} \rightarrow r_{val}, S, \Longrightarrow, d')$

- Suppose $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$,
  where $v \in V$ and $\mathbf{E}_=(v) = r \neq \bot$
  Now $\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d') = \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$. But $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$ by the inductive hypothesis. Hence $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d')$.
- Otherwise, suppose
  $$l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \bigcup\{\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) | v \rightarrow r \in \mathbf{E}_\rightarrow\}$$
  where $v \in V$ and $\exists v | v \rightarrow r \in \mathbf{E}_\rightarrow$
  $$\mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d') = \bigcup\{\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d') | v \rightarrow r \in \mathbf{E}_\rightarrow\}$$
  But, by the inductive hypothesis,
  $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ implies $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d')$ for all r in the productions for v in $\mathbf{E}$. Hence $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d')$.
- Otherwise, if $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top_M$ for some $v \in V_m$,
  then $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d') = \top_M$.
- Otherwise, if $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top$ for some $v \in V_p$,
  then $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d') = \top$.
- In the case of all other forms for l, $\mathcal{L}(\mathbf{E}, r_1 \cup r_2, S, \Longrightarrow, d)$, $\mathcal{L}(\mathbf{E}, r_1 r_2, S, \Longrightarrow, d)$ and $\mathcal{L}(\mathbf{E}, r_1/r_2, S, \Longrightarrow, d)$, $\mathcal{L}(\mathbf{E}, \{r\}, S, \Longrightarrow, d)$ the result is immediate from the inductive hypothesis and the monotonicity of union, product, quotient and Kleene-closure.

Proposition 4.2.2 (Stability of h) *Let $\mathbf{E}$ be a type-environment, $r \in R$ and let $S = (s, e, c, b, h)$ be a machine-state. Let $\Longrightarrow$ be a legal-reachability relation. Let $h_{new} = h[a_{new} \mapsto (a_0 \ldots a_n)]$ be a heap, where $a_{new} \notin dom(h)$.*
Let $S_{new} = (s, e, c, b, h_{new})$. Then:

$$\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$$

.

Proof: Let us note first that

$$dom(h) \subseteq dom(h_{new}) \tag{6}$$

and that, if $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ then

$$h(a) = h_{new}(a) \tag{7}$$

Consider $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.
We proceed by induction on the derivation-length of $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.
Base case: If the derivation of $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ is of depth 0, then

- $\mathcal{L}(\mathbf{E}, \emptyset, S, \Longrightarrow, d) = \emptyset$ so l cannot be derived from this.
- If $l \in \mathcal{L}(\mathbf{E}, \mathbf{1}, S, \Longrightarrow, d) = \{\epsilon\}$ then $l \in \mathcal{L}(\mathbf{E}, \mathbf{1}, S, \Longrightarrow, d')$.
- If $l = \hat{a} \in \mathcal{L}(\mathbf{E}, S(a), S, \Longrightarrow, d) = \{\hat{a}\}$ then, by inequality (6), $\hat{a} = \mathcal{L}(\mathbf{E}, S(a), S_{new}, \Longrightarrow, d)$
- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a record-key of arity n. Now $\mathcal{L}(\mathbf{E}, D(a_{key}), S_{new}, \Longrightarrow, d)$
$= \{\hat{a'} | h_{new}(a') = (a_{key}, a'_1 \ldots a'_n) \text{ for some } a'_1 \ldots a'_n \in dom(h_{new})\}$
But $h_{new}(a) = h(a)$, and $h(a) = (a_{key}, a_1 \ldots a_n)$,
where $a_1, \ldots a_n \in dom(h) \subseteq dom(h_{new})$.
Hence $\hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}), S_{new}, \Longrightarrow, d)$.

Inductive step: Suppose that $\mathbf{E}$ is a type-environment, $r \in R$, and let h be a heap. Let $h_{new} = h[a' \mapsto (a_0 \ldots a_n)]$ be a heap, where $a' \notin dom(h)$. Then $\hat{a} \in \mathcal{L}(\mathbf{E}, r, h)$ implies that $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$ provided that the derivation of $\hat{a} \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ is of depth $\leq \delta$.
Now consider a derivation of $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ of depth $\delta + 1$:

- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a record-key of arity n, and $r_1 \ldots r_n \in R$ are type-expressions for which $L_i = \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d)$ are manifest monologs,

$$\mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S_{new}, \Longrightarrow, d)$$
$$= \{\hat{a'} | h_{new}(a') = (a_{key}, a'_1 \ldots a'_n), \hat{a'_i} \in L'_i)\}$$

where $L'_i = \mathcal{L}(\mathbf{E}, r_i, S_{new}, \Longrightarrow, d)$
But $\hat{a} \in \{\hat{a'} | h(a') = (a_{key}, a_1 \ldots a_n), \hat{a}_i \in L_i)\}$
Thus $h(a) = h_{new}(a) = (a_{key}, a_1 \ldots a_n)$, $a_i \in \mathcal{L}(\mathbf{E}, r_i, S, \Longrightarrow, d)$.
But, by the inductive hypothesis, $a_i \in \mathcal{L}(\mathbf{E}, r_i, S_{new}, \Longrightarrow, d)$;
hence $\hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r_1, \ldots r_n), S_{new}, \Longrightarrow, d)$.
- Consider $l = \hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r), S, \Longrightarrow, d)$ where $a_{key} \in dom(h)$ is a vector-key, and $r \in R$. Now

$$\mathcal{L}(\mathbf{E}, D(a_{key}, r), S_{new}, \Longrightarrow, d)$$
$$= \{\hat{a'} | h_{new}(a') = (a_{key}, a_1, \ldots a_n), (a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)\}$$

But $\hat{a} \in \{\hat{a'} | h(a') = (a_{key}, a_1, \ldots a_n), (a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)\}$
so $h(a) = h_{new}(a) = (a_{key}, a_1, \ldots a_n)$ where $(a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$.
Hence, by the inductive hypothesis, $(a_1 \ldots a_n) \in \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$.
Thus we see that $\hat{a} \in \mathcal{L}(\mathbf{E}, D(a_{key}, r), S_{new}, \Longrightarrow, d)$.

- If $\mathcal{L}(\mathbf{E}, D(a', r_1 \ldots r_n), S, \Longrightarrow, d) = \emptyset$, $\hat{a} \notin \emptyset$, hence this case is covered by *ex falsa libet.*

- If $l = \hat{a} \in \mathcal{L}(\mathbf{E}, r_{arg} \to r_{val}, S, \Longrightarrow, d)$ where $r_1, r_2 \in R$,
  $\mathcal{L}(\mathbf{E}, r_{arg} \to r_{val}, S_{new}, \Longrightarrow, d) =$
  $\{\hat{a'} | h_{new}(a') = (a_{ProcKey}, \ldots)$
  and $\forall S' \; \forall d_1 \leq d . S_{new} \Longrightarrow S', l_{arg} \in \mathcal{L}(\mathbf{E}, r_{arg}, S', \Longrightarrow, d_1)$ implies
  $l_{val} \in \mathcal{L}(\mathbf{E}, r_{val}, S'', \Longrightarrow, d_1)$
  where $S' = (l_{val}, \mathbf{e}', c', b', h')$
  and $obey(l_{arg}, \mathbf{e}', h'(a'), h', d_1) = (l_{val}, \mathbf{e}_{val}, h'_{val}, \mathbf{S})$
  and $S'' = (l_{val}, \mathbf{e}_{val}, c', b', h'_{val})\}$
  Now $h(a) = (a_{ProcKey}, \ldots)$
  and $\forall S' \; \forall d_1 \leq d$ if $S \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$
  then $l_{arg} \in \mathcal{L}(\mathbf{E}, r_{arg}, S', \Longrightarrow, d_1)$ implies $l_{val} \in \mathcal{L}(\mathbf{E}, r_{val}, S'', \Longrightarrow, d_1)$
  where $obey(l_{arg}, \mathbf{e}', h'(a), h', d_1) = (l''_{val}, \mathbf{e}'', h'', \mathbf{S})$ and $S'' = (l_{val}, \mathbf{e}'', c', b', h'')$
  Consider S_1 where $S_{new} \Longrightarrow S_1$. Then since $S \Longrightarrow S_{new}$, and $\Longrightarrow$ is transitive, then $S \Longrightarrow S_1$.
  So we conclude that $\hat{a} \in \mathcal{L}(\mathbf{E}, r_{arg} \to r_{val}, S_{new}, \Longrightarrow, d)$

  (Note that this case depends on the fact that the type of the code-block is stable with respect to $\Longrightarrow$ and *not* on the inductive hypothesis).

- Suppose $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$,
  where $v \in V$ and $\mathbf{E}_{=}(v) = r \neq \perp$
  Now $\mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$ But $l \in \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$ by the inductive hypothesis. hence $l \in \mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d)$.

- Otherwise, suppose
  $$l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \bigcup \{\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) | v \to r \in \mathbf{E}_{\to}\}$$
  where $v \in V$ and $\exists v | v \to r \in \mathbf{E}_{\to}$
  $$\mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d) = \bigcup \{\mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d) | v \to r \in \mathbf{E}_{\to}\}$$
  But, by the inductive hypothesis,
  $l \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$ implies $l \in \mathcal{L}(\mathbf{E}, r, S_{new}, \Longrightarrow, d)$
  for all r in the productions for v in $\mathbf{E}$. Hence $l \in \mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d)$.

- Otherwise, if $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top_M$ for some $v \in V_m$,
  then $l \in \mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d) = \top_M$.

- Otherwise, if $l \in \mathcal{L}(\mathbf{E}, v, S, \Longrightarrow, d) = \top$ for some $v \in V_p$,
  then $l \in \mathcal{L}(\mathbf{E}, v, S_{new}, \Longrightarrow, d) = \top$.

- In the case of all other forms for l, $\mathcal{L}(\mathbf{E}, r_1 \cup r_2, S, \Longrightarrow, d)$, $\mathcal{L}(\mathbf{E}, r_1 r_2, S, \Longrightarrow, d)$ and $\mathcal{L}(\mathbf{E}, r_1 / r_2, S, \Longrightarrow, d)$, $\mathcal{L}(\mathbf{E}, \{r\}, S, \Longrightarrow, d)$ the result is immediate from the inductive hypothesis and the monotonicity of union, product, quotient and Kleene-closure.

4.3 Primitives: Constructors and Selectors

Definition 4.3.1 (Record Constructor) *A primitive p is said to be a* record constructor *for a key-object a_{key} of arity n if*

$$obey(s\hat{a_1}\dots\hat{a_n}, \mathbf{e}, p, h, d) = (sa, \mathbf{e}, h[a \mapsto (a_{key}, a_1 \dots a_n)], \epsilon)$$

where a is an address which does not occur in $dom(h)$. Moreover, if $h(a_{con}) = (a_{ProcKey}, p)$ then a_{con} is said to be a record-constructor object in the heap h for the key a_{key}.

Proposition 4.3.1 (Constructor Type) *Let a_{con} be a record constructor object for a_{key} in a heap h, and $\Longrightarrow$ be a non-mutating legal-reachability relation. Let $S = (s, \mathbf{e}, c, b, h)$ be a VM-state. Let $v_i \in V_m$ be monolog variables. Then:*

$$\widehat{a_{con}} \in \mathcal{L}(\mathbf{E}, v_1 v_2 \dots v_n \to D(a_{key} v_1, \dots v_n), S, \Longrightarrow, d)$$

Proof: We have to show that the conditions specified in equation (4) hold. Let $S \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$. Let $d \geq 0$ be an integer and let $d_1 \leq d$. Since $\Longrightarrow$ is non-mutating,
$h'(a_{con}) = h(a_{con}) = (a_{ProcKey}, p)$, say.

Let $s' \in \mathcal{L}(\mathbf{E}, v_1 v_2 \dots v_n, S', \Longrightarrow, d_1)$ then $s' = \hat{a'_1} \dots \hat{a'_n}$,
where $a'_i \in \mathcal{L}(\mathbf{E}, v_i, S', \Longrightarrow, d_1) = L'_i$, say.

Now $obey(s', \mathbf{e}', p, h', d_1) = obey(\hat{a'_1} \dots \hat{a'_n}, \mathbf{e}, p, h', d_1)$
$= (\hat{a''}, \mathbf{e}', h'[a'' \mapsto (a_{key}, a'_1 \dots a'_n)], \epsilon) = (\hat{a''}, \mathbf{e}', h'', \epsilon)$
where a'' is an address which does not occur in $dom(h')$. But

$$\mathcal{L}(\mathbf{E}, D(a_{key}, v_1, \dots v_n), S'', \Longrightarrow, d_1) = \{\hat{a} | h''(a) = (a_{key}, a''_1 \dots a''_n), \hat{a''_i} \in L''_i)\}$$

where $L''_i = \mathcal{L}(\mathbf{E}, v_i, S'', \Longrightarrow, d_1)$.

But $L'_i \subseteq L''_i$ by proposition 4.2.2.
Hence $a'' \in \mathcal{L}(\mathbf{E}, D(a_{key}, v_1, \dots v_n), S'', \Longrightarrow, d_1)$ Hence result.

Note that in the above proof, we say nothing about whether the v_i are distinct or unbound by type-equations or productions. The most general type for the constructor arises from the case when they are all distinct and unbound.

Definition 4.3.2 (Selector) *A primitive p is said to be the i'th* selector *for a record-key-object a_{key} of arity $n \geq i$ if $h(a) = (a_{key}, a_1 \dots a_n)$ implies*

$$obey(s\hat{a}, \mathbf{e}, (a_{ProcKey}, p), h, d) = (s\hat{a_i}, \mathbf{e}, h, \mathbf{S})$$

Moreover, if $h(a_{sel}) = (a_{ProcKey}, p)$ then a_{sel} is said to be the ith record-selector object in the heap h for the key a_{key}.

Proposition 4.3.2 (Selector Type) *If a_{sel} is the i'th record-selector object for a record-key a_{key} of arity $n \geq i$ in a heap h, and $S = (s, \mathbf{e}, c, b, h)$ is a VM-state and $\Longrightarrow$ is a non-mutating legal-reachability relation then*

$$\widehat{a_{sel}} \in \mathcal{L}(\mathbf{E}, D(a_{key} v_1, \dots v_n \to v_i, S, \Longrightarrow, d)$$

Proof: Let $S \Longrightarrow S' = (s', \mathbf{e}', c', b', h')$. Let $d \geq 0$ be an integer, and let $d_1 \leq d$. Since $\Longrightarrow$ is non-mutating, $h'(a_{sel}) = h(a_{sel}) = (a_{ProcKey}, p)$, say. Let $\hat{a'} \in \mathcal{L}(\mathbf{E}, D(a_{key}, v_1, \ldots v_n), S', \Longrightarrow, d)$ then $h'(a') = (a_{key}, a_1 \ldots a_n)$ where $\hat{a}_i \in \mathcal{L}(\mathbf{E}, v_i, S', \Longrightarrow, d) = L_i$, say.

Now $S'' = obey(\hat{a'}, \mathbf{e}, p, h', d) = (\hat{a}_i, \mathbf{e}, h', \ldots)$ and $\hat{a}_i \in L_i$. But, by proposition 4.2.2, $L_i \subseteq \mathcal{L}(\mathbf{E}, v_i, S'', \Longrightarrow, d)$. Hence result.

4.4 Terms and Recognisers

In the previous section we specified that the $\mathbf{E}_?$ component of a type-environment describes identifiers bound to objects which are able to *recognise* members of a particular data-class. These are required because we sometimes need to be able to distinguish between objects which belong to a type-union. In order to be able to make use of such *recognisers* we need to keep a limited representation of the top of stack. To this end we introduce *terms.* A term can be either

1. An address in $dom(h)$
2. $"w"$, where $w \in dom(h)$ is an identifier.
3. w, where $w \in dom(h)$ is an identifier.
4. $w(T)$, where $w \in dom(h)$ is an identifier.
5. $\perp_T$.

Definition 4.4.1 (Term-evaluation) *Let T be a term. Let $S = (s, \mathbf{e}, c, b, h)$ be a VM state. Then the value $\mathcal{V}(T, S)$ of T in the state S is defined as follows.*

1. *If $T = a \in dom(h)$ then $\mathcal{V}(T, S) = a$.*
2. *If $T = "w"$, where w is an identifier, then $\mathcal{V}(T, S) = w$*
3. *If $T = w$, where w is an identifier, then $\mathcal{V}(T, S) = \mathbf{e} \otimes w$*
4. *If $T = w(T')$, where w is an identifier, then let $a' = \mathcal{V}(T', S)$. Then $\mathcal{V}(T, S) = a''$ where $obey(\hat{a'}, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (\hat{a''}, \mathbf{e}'', h'', \ldots)$ Othewise $\mathcal{V}(T, S) = \mathtt{u}$*
5. *If $T = \perp_T$, then $\mathcal{V}(T, S) = \mathtt{u}$.*

4.5 Describing Machine States

We have seen how a type-environment allows us to give meaning to an expression of an ERWA as denoting a language whose alphabet is objects of the VM. In this section we see how the $\mathbf{E}_:$ and $\mathbf{E}_?$ components of a type-environment are used to describe a state of the VM, in conjunction with a grammar describing the stack, and a term which may evaluate to the top of stack.

Definition 4.5.1 (State-description) *Let $\mathbf{E}$ be a type-environment, let $S = (s, \mathbf{e}, c, b, h)$ be a VM-state, and $\Longrightarrow$ be a legal-reachability relation. Let $d \geq 0$ be an integer, $r \in R$ and term T We say that $(r, T, \mathbf{E})$ d-describes $S = (s, \mathbf{e}, b, h)$, and write $(r, T, \mathbf{E}) \vdash_d S$, if the following conditions are satisfied:*

1. $s \in \mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d)$
2. If $T \neq \bot_T$ then $s = s'\hat{n}$ where $\mathcal{V}(T, S) = n$
3. If $w \in dom(\mathbf{E}_{:})$ and $\mathbf{e} \otimes w \neq \mathtt{u}$ then $\mathbf{e} \otimes w \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{:}(w), S, \Longrightarrow, d)$
4. If $w \in dom(\mathbf{E}_{?})$ then
   - $\mathbf{e} \otimes w \in \mathcal{L}(\mathbf{E}, \top \rightarrow \mathtt{Bool}, S, \Longrightarrow, d)$
   - If $s = s'\hat{a}$ then if $a \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{?}(w), S, \Longrightarrow, d)$ then
   $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\mathtt{t}, \mathbf{e}, h, \ldots)$$
   otherwise
   $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\mathtt{f}, \mathbf{e}, h, \ldots)$$
5. If w is bound in $\mathbf{e}$ then $w \in dom(\mathbf{E}_{:}) \cup dom(\mathbf{E}_{?})$

Note that an occurrence of w in $\mathbf{e}$ which is not in scope is not given a type in $\mathbf{E}$.

Definition 4.5.2 (State-description (Universal)) *If* $\mathbf{E}$ *is a type-environment and* S *is a state, for which*

$$(\top, \bot_T, \mathbf{E}) \vdash_d S$$

for all $d \geq 0$ *then we write* $\mathbf{E} \vdash S$.

4.6 Annotations say what each instruction does.

Suppose we have a type-environment $\mathbf{E}$, a machine state $S = (s, \mathbf{e}, c, b, h)$ where $(\top, \bot_T, \mathbf{E}) \vdash_d S$. A *compiler* will extend the environment and heap of S, giving rise to S'; it will make new objects on the heap and will bind these to identifiers in the environment[2]. The question we address in this section is, "how can we determine that a new type environment $\mathbf{E}'$, derived from $\mathbf{E}$, describes S'?".

In order to validate $\mathbf{E}'$, we introduce the idea of an *annotation* of a code-block, which will allow us to characterise the execution of the new code-blocks on a step-by-step basis. With an annotation of a code-block we introduce a *local type environment* which allows us to assign independent types to the local variables and labels of that block.

Note that we cannot necessarily perform the validation of the addition of the new bindings on a one-at-a-time basis because they may be mutually recursive.

We first need to be able to assign a type to a literal occurring in a *pushq* instruction. Assigning a type to a literal is problematic. The type which throws away the least information about the literal is the singleton type $S(h)$. This turns out to be too restrictive in our type-inference scheme for local variables[3]. The definition chosen, in a Poplog VM context, represents the best compromise.

[2]The compiler can be incorporated in the VM, as is in fact done in the Poplog VM. However our type-analysis is performed regarding the compiler as the kind of *deus ex machina* referred to.

[3]Type inference is not covered in this report

Instr	c_{next}	Stack	Term	Cond
$pushq(n)$	$c+1$	$r\bar{n}$	n	
$push(w)$	$c+1$	$r\mathbf{E}'_:(w)$	w	
$pop(w)$	$c+1$	$r/\mathbf{E}'_:(w)$	$\bot_T$	
$call(w)$	$c+1$	$(r/r_{arg1})r_{val1}$	$w(T)$	where $\mathbf{E}'_:(w) = r_{arg1} \rightarrow r_{val1}$
				r_{arg1}, r_{val1} are manifest monologs
$call(w)$	$c+1$	$(r/r_{arg1})r_{val1}$	$\bot_T$	where $\mathbf{E}'_:(w) = r_{arg1} \rightarrow r_{val1}$
$label(v)$	$c+1$	v	$\bot_T$	$(v \rightarrow r) \in \mathbf{E}_\rightarrow$
$goto(v)$	$c'+1$	v	T	$b(c') = label(v)$
				$(v \rightarrow r) \in \mathbf{E}_\rightarrow)$
$ifnot(v)$	$c'+1$	v	$\bot_T$	$b(c') = label(v)$
				$(v \rightarrow r/\top_M) \in \mathbf{E}_\rightarrow$
	$c+1$	$r/\top_M$	$\bot_T$	

Table 1: Annotating a Code Block

Definition 4.6.1 (The type of a literal) *If $a \in dom(h)$ is an address for which $h(a) = (a_{key} \ldots)$, then the type of the literal a is $\bar{a} = D(a_{key})$*

Definition 4.6.2 (Annotation) *Let b be a code block of n instructions. Let $\mathbf{E}$, $\mathbf{E}'$ be type-environments. We say that a sequence $\mathbf{B}$ for which $\mathbf{B}_c = (r_c, T_c)$, where $r_c \in R$ and T_c is a term, is an annotation of b in $\mathbf{E}'$ with specification $r_{arg} \rightarrow r_{val}$ and conformant with $\mathbf{E}$ if the following conditions hold:*

1. If $b_c = label(v)$, for some integer c, then $v \in V_p$.
2. $\mathbf{E}'_:(w) = \mathbf{E}_:(w)$ if w is not a local variable of b.
3. $\mathbf{E}'_:(w) \neq \bot$ if w is a local variable of b.
4. $\mathbf{E}'_?(w) = \mathbf{E}_?(w)$
5. $\mathbf{E}'_=(w) = \mathbf{E}_=(w)$ for all w occurring in $\mathbf{E}_:$, $dom(\mathbf{E}_=)$, $range(\mathbf{E}_=)$, $\mathbf{E}_\rightarrow$, r_{arg}, r_{val}.
6. $\mathbf{E}_\rightarrow(w) = \emptyset$.
7. $\mathbf{B}_0 = (r_{arg}, \bot_T)$
8. If $\mathbf{B}_c = (r, T)$ then its successor(s) are annotated according to Table 1. In the table, the possible next values c are given in the column labelled "c_{next}". the columns labelled 'Stack' and 'Term' are the value of $\mathbf{B}_{c_{next}}$.

We write $annotates(B, b, r_{arg} \rightarrow r_{val}, \mathbf{E}, \mathbf{E}')$.

The main proposition of this section shows that an annnotation of a code block determines the type of the code-block. That is to say, if the stack is described by r on entry, then it will be described by $(r/r_{arg})r_{val}$ on exit, provided the language denoted by r is exactly divisible by that denoted r_{arg}. However, one code block may contain calls to other code blocks, possibly including itself, so our main proposition applies to collections of code-blocks. However, for the nonce, let us consider the execution of a single code-block.

Lemma 4.6.1 (Stability over lvars) *If* $\mathbf{E}$ *and* $\mathbf{E}'$ *are type-environments for which* $\mathbf{E}_{\rightarrow} = \mathbf{E}'_{\rightarrow}$ *and* $r \in R$, S *is a machine state,* $\Longrightarrow$ *is a legal reachability relation and* $d \geq 0$ *is an integer depth for which* $\mathbf{E}_{=}(w) = \mathbf{E}'_{=}(w)$ *for all* w *occurring in* r, $dom(\mathbf{E}_{=})$, $range(\mathbf{E}_{=})$, $\mathbf{E}_{\rightarrow}$, *and* $S = (s, \mathbf{e}, c, b, h)$ *is a machine-state and* $S' = (s, lvars(\mathbf{e}), c, b, h)$ *then*

$$\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', r, S', \Longrightarrow, d)$$

Proof: By lemma 4.2.1 $\mathcal{L}(\mathbf{E}, r, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', r, S, \Longrightarrow, d)$,
$= \mathcal{L}(\mathbf{E}', r, S', \Longrightarrow, d)$, by lemma 4.2.2.

Proposition 4.6.1 (Correctness of Annotation) *Let* b *be a code block of* n *instructions for which* $pop(w)$ *only occurs for local variables. Let* $\mathbf{E}$, $\mathbf{E}'$ *be type-environments. Let* $d > 0$ *be an integer. Let* $S = (s, \mathbf{e}, c, b, h)$ *be a VM-state. Suppose:*

$$annotates(\mathbf{B}, b, r_{arg} \rightarrow r_{val}, \mathbf{E}, \mathbf{E}')$$

And suppose $s \in \mathcal{L}(\mathbf{E}, r_{arg}, S, \Longrightarrow, d)$.
Let $(s', \mathbf{e}', h', (S_1 \ldots S_m)) = obey(s, \mathbf{e}, b, h, d)$, *and suppose that the divisions of table 1 are exact for each* S_j. *Suppose that, for some d:*

$$(\top, \bot_T, \mathbf{E}) \vdash_d S \tag{8}$$

Then

$$(\mathbf{B}_c, \mathbf{E}') \vdash_d S_j$$

where c *is the index of the instrution in* b *involved in deriving* S_j.

Proof: We proceed by induction on j, and have to show that the conditions of definition 4.5.1 hold, both for the base case $j = 0$ and for the inductive step.

Base Case:

$S_0 = (s, lvars(\mathbf{e}, b), b, h)$ from equation 3 and $\mathbf{B}_0 = (r_{arg}, \bot_T)$ from definition 4.6.2, item 6.

1. We know that $s \in \mathcal{L}(\mathbf{E}, r_{arg}, S, \Longrightarrow, d)$. and so by lemma 4.6.1
   $s \in \mathcal{L}(\mathbf{E}', r_{arg}, S_0, \Longrightarrow, d)$,
2. The term is undefined for $\mathbf{B}_0$, so there is nothing to prove.
3. Consider $w \in dom(\mathbf{E}'_{:})$. Then

- Either w is a local variable of b, in which case $\mathbf{e} \otimes w = \mathtt{u}$, so there is nothing to prove.
- Or $\mathbf{E}'_{:}(w) = \mathbf{E}_{:}(w)$ If $\mathbf{e} \otimes w \neq \mathtt{u}$ then $\mathbf{e} \otimes w \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{:}(w), S, \Longrightarrow, d)$ $= \mathcal{L}(\mathbf{E}', \mathbf{E}'_{:}(w), S_0, \Longrightarrow, d)$, by lemma 4.6.1

4. Consider $w \in dom(\mathbf{E}'_{?})$. Now $\mathbf{E}'_{?} = \mathbf{E}_{?}$

   - $\mathbf{e} \otimes w \in \mathcal{L}(\mathbf{E}, \top \rightarrow \mathtt{Bool}, S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', \top \rightarrow \mathtt{Bool}, S_0, \Longrightarrow, d)$, by lemma 4.6.1.
   - If $s = s'\hat{a}$ then if $a \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{?}(w), S, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', \mathbf{E}'_{?}(w), S, \Longrightarrow, d)$ then

     $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\mathtt{t}, \mathbf{e}, h, \ldots)$$

     otherwise

     $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\mathtt{f}, \mathbf{e}, h, \ldots)$$

   - If w is bound in $lvars(\mathbf{e}, b)$ then w is bound in $\mathbf{e}$ (for we regard uninitialised locals as unbound).

Hence $(\mathbf{B}_0, \mathbf{E}) \vdash_d S_0$

Inductive Step: There are 5 clauses to the definition of $\vdash$, and 7 kinds of instruction for the VM, so there are 35 possible combinations to consider. Fortunately, not all of these combinations involve instructions which change the machine environment, and we are not allowing local recognisers, so life is somewhat simpler.

Consider $S_{j+1} = exec_d(S_j)$. Let $S_j = (s, \mathbf{e}, c, b, h)$. We show, on a case-by-case basis, that if $\mathbf{B}_c \vdash S_j$ then $\mathbf{B}_{c'} \vdash S_{j+1}$. for each c' which is an appropriate successor value of c. Let $\mathbf{B}_c = (r, T)$

Let $L_s = \mathcal{L}(\mathbf{E}, r, S_j, \Longrightarrow, d)$, so that $s \in L_s$.

1. If $b_c = pushq(n)$ then $exec_d(s, \mathbf{e}, c, b, h) = (s\hat{n}, \mathbf{e}, c{+}1, b, h)$ and $\mathbf{B}_{c+1} = (r\bar{n}, n)$. But $s \in L_s$, $n \in \mathcal{L}(\mathbf{E}', \bar{n}, S_j, \Longrightarrow, d)$ and so $s\hat{n} \in \mathcal{L}(\mathbf{E}', r\bar{n}, S_j, \Longrightarrow, d)$. $=$ $\mathcal{L}(\mathbf{E}', r\bar{n}, S_{j+1}, \Longrightarrow, d)$, since $\mathbf{e}$ is unchanged.

   Considering now the term component, n, we see that $\mathcal{V}(\mathbf{e}, n) = n$ and so is equal to the top of stack.

   Hence, since $\mathbf{e}$ is unchanged, $(\mathbf{E}', \mathbf{B}_{c+1}) \vdash S_{j+1}$

2. If $b_c = push(w)$ then let $a = \mathbf{e} \otimes w$.
   Then $exec_d(s, \mathbf{e}, c, b, h) = (s\hat{a}, \mathbf{e}, c + 1, b, h) = S_{j+1}$
   and $\mathbf{B}_{c+1} = (r\mathbf{E}'_{:}(w), w)$

   But $s \in L_s$ and, by the inductive hypothesis,
   $\hat{a} \in \mathcal{L}(\mathbf{E}', \mathbf{E}'_{:}(w), S_j, \Longrightarrow, d) = L_w$,say.
   So $s\hat{a} \in L_s L_w = \mathcal{L}(\mathbf{E}', r\mathbf{E}'_{:}(w), S_j, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', r\mathbf{E}'_{:}(w), S_{j+1}, \Longrightarrow, d)$, by lemma 4.2.2.

   Considering now the term component, w, we see that $\mathcal{V}(\mathbf{e}, w) = a$
   Hence, since no environmental changes occur, $(\mathbf{E}, \mathbf{B}_{c+1}) \vdash S_{j+1}$

3. If $b_c = pop(w)$ then $B_{c+1} = (r/\mathbf{E}'_{:}(w), \bot_T)$.
   Since the monolog $M_w = \mathcal{L}(\mathbf{E}', \mathbf{E}'_{:}(w), S_j, \Longrightarrow, d)$ divides L_s exactly (by the conditions of the proposition) $s = s'\hat{n}$, where $\hat{n} \in M_w$.
   Let $\mathbf{e}' = \mathbf{e}[w \mapsto \hat{n}]$, then $exec_d(s, \mathbf{e}, c, b, h) = (s', \mathbf{e}', c, b, h)$.
   Considering the conditions for $(\mathbf{E}', \mathbf{B}_{c+1}) \vdash S_{j+1}$
   (a) $s'\hat{n} \in L_s$, $\hat{n} \in M_w$, hence $s' \in L_s/M_w = \mathcal{L}(\mathbf{E}', r/\mathbf{E}'_{:}(w), S_j, \Longrightarrow, d) = \mathcal{L}(\mathbf{E}', r/\mathbf{E}'_{:}(w), S_{j+1}, \Longrightarrow, d)$, by lemma 4.2.2 recalling that w is local by conditions of theorem, so $\mathbf{e}_0 = \mathbf{e}'_0$.
   (b) The term of B_{c+1} is undefined. Hence there is nothing to prove.
   (c) Consider $w' \in dom(\mathbf{E}'_{:}(w))$. If $w' = w$ then $\mathbf{e}' \otimes w = \hat{n} \in M_w$. Otherwise $\mathbf{e}' \otimes w = \mathbf{e} \otimes w$.
   In either case $\mathbf{e}' \otimes w \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{:}(w), S_j, \Longrightarrow, d)$.
   Hence $\mathcal{L}(\mathbf{e}' \otimes w \in \mathbf{E}, \mathbf{E}_{:}(w), S_{j+1}, \Longrightarrow, d)$ by lemma 4.2.2.
   (d) Consider $w' \in dom(\mathbf{E}'_{?}(w))$.
   - $\mathbf{e}' \otimes w \in \mathcal{L}(\mathbf{E}, \top \to \texttt{Bool}, S_{j+1}, \Longrightarrow, d)$ by a similar argument to the above.
   - If $s = s'\hat{a}$ then if $a \in \mathcal{L}(\mathbf{E}, \mathbf{E}_{?}(w), S_j, \Longrightarrow, d)$ then
   $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\texttt{t}, \mathbf{e}, h, \ldots)$$
   hence by lemma 3.2.2
   $$obey(s, \mathbf{e}', \mathbf{e} \otimes w, h, d) = (s'\texttt{t}, \mathbf{e}'', h, \ldots)$$
   otherwise
   $$obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\texttt{f}, \mathbf{e}, h, \ldots)$$
   (e) We have ensured that all variables are given a type in $\mathbf{E}'$ by the definition of annotation.

   Hence $(B_{c+1}, \mathbf{E}) \vdash S_{j+1}$
4. If $b_c = call(w)$ then
   $exec_d(s, \mathbf{e}, c, b, h) = (s', \mathbf{e}', c+1, b, h)$
   where $(s', \mathbf{e}', h', \mathbf{S}) = obey(s, \mathbf{e}, \mathbf{e} \otimes w, h, d-1)$
   There are the following cases:
   (a) If $\mathbf{E}'_{:}(w) = r_{arg1} \to r_{val1}$ where r_{arg1} r_{val1} are not both manifest monologs.
   $\mathbf{B}_{c+1} = ((r/r_{arg1})r_{val1}, \bot_T)$
   Now, since division is exact, $s = s_0 s_{arg1}$
   where $s_{arg1} \in \mathcal{L}(\mathbf{E}, r_{arg1}, S_j, \Longrightarrow, d)$, by condition 8.
   Hence, by equation 4, if
   $$obey(s_{arg1}, \mathbf{e}, \mathbf{e} \otimes w, h, d-1) = (s_{val1}, \mathbf{e}'', h'', \ldots)$$
   where $s_{val1} \in \mathcal{L}(\mathbf{E}, r_{val1}, S_j, \Longrightarrow, d)$. Hence by lemma 3.2.1
   $$s' = s_0 s_{val1} \in \mathcal{L}(\mathbf{E}, (r/r_{arg1})r_{val1}, S_j, \Longrightarrow, d)$$
   So $(\mathbf{E}', B_{c+1}) \vdash_d S_{j+1}$

(b) r_{arg1}, r_{val1} are manifest monologs. $\mathbf{B}_{c+1} = (r/r_{arg1})r_{val1}\ ,\ w(T))$ This is similar to the previous case, except for the term. Since division is exact and r_{arg1} is a manifest monolog, $s = s'\hat{n}$. where $n = \mathcal{V}(T, S_j)$.
Now $\mathcal{V}(w(T), S) = a''$ where $(\hat{a''}, \mathbf{e}'', h'', \mathbf{S}) = obey(\hat{a'}, \mathbf{e}, \mathbf{e} \otimes w, h, d-1)$
Hence, by lemma3.2.1 $obey(s'\hat{a'}, \mathbf{e}, \mathbf{e} \otimes w, h, d) = (s'\hat{a''}, \mathbf{e}'', h'', \mathbf{S}'')$ That is a'' is the top of stack when we obey the instruction.
Hence $(\mathbf{E}, B_{c+1}) \vdash_d S_{j+1}$

Hence $(B_{c+1}, \mathbf{E}) \vdash_d S_{j+1}$

5. If $b_c = label(v)$ then $exec_d(s, \mathbf{e}, c, b, h) = (s, \mathbf{e}, c+1, b, h)$
and $\mathbf{B}_{c+1} = (v, T)$, $v \to r \in \mathbf{E}_{i\to}$.
But $s \in L_s$ and $L_s = \mathcal{L}(\mathbf{E}, r, S_j, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, v, S_j, \Longrightarrow, d)$ by equation (5) so $(B_{c+1}, \mathbf{E}) \vdash_d S_{j+1}$

6. If $b_c = goto(v)$, then let $b(c') = label(v)$.
Now $exec_d(s, \mathbf{e}, c, b, h) = (s, \mathbf{e}, c'+1, b, h)$
and $\mathbf{B}_{c'+1} = (v, T)$, $v \to r \in \mathbf{E}_{i\to}$.
But $s \in L_s$. and $L_s = \mathcal{L}(\mathbf{E}, r, S_j, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, v, S_j, \Longrightarrow, d)$ by (5). Hence $(B_{c'+1}, \mathbf{E}) \vdash_d S_{j+1}$

7. If $b_c = ifnot(v)$ then let $b(c') = label(v)$.

$\mathbf{B}_{c+1} = (r/\top_M, \bot_T)$, $\mathbf{B}_{c'+1} = (v, \bot_T)$, $v \to r/\top_M \in \mathbf{E}_{i\to}$.
The condition that the division is exact means that only two of the possible cases in Table 1 can occur. We can write $s = s'\hat{n}$, and, since $\top_M$ is a monolog, $s' \in L_s/\top_M$.

(a) $n = \mathtt{f}$. Here $exec_d(s, \mathbf{e}, c, b, h) = (s', \mathbf{e}, c'+1, b, h)$
But $s \in L_s/\top_M$
and $L_s/\top_M = \mathcal{L}(\mathbf{E}, r/\top_M, S_j, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, v, S_j, \Longrightarrow, d)$ by (5).
Hence $(B_{c'+1}, \mathbf{E}) \vdash_d S_{j+1}$

(b) Otherwise, $exec_d(s, \mathbf{e}, c, b, h) = (s', \mathbf{e}, c+1, b, h))$.
But $s \in L_s/\top_M$
and $L_s/\top_M = \mathcal{L}(\mathbf{E}, r/\top_M, S_j, \Longrightarrow, d) \subseteq \mathcal{L}(\mathbf{E}, v, S_j, \Longrightarrow, d)$ by (5).
Hence $(B_{c+1}, \mathbf{E}) \vdash_d S_{j+1}$.

5 Discussion

This report has pointed a way towards using the theory of formal languages can to describe data-types occurring in a stack-machine. Our main result has been to show that an annotation of a code-block accurately describes the behaviour of the machine provided the divisions are always exact.

However, an annotation is tedious to construct, and we may not be able to tell immediately whether the divisions *are* always exact. While I shall not pursue the formal investigation of this problem further in this report, I would like to indicate that it is fairly clear what must be done to get a practical treatment of type along this approach. The results of section 2 give us a basis of *simplifying* annotations. Such simplification can leave all expressions of R occurring in the annotation *quotient-free*, thus assuring that divisions are exact. For example, if `Int` is a manifest monolog, then $\mathcal{L}(\mathbf{E}, \texttt{Int/Int}, S, \Longrightarrow, d) = \mathbf{1}$ for any type-environment, state and legal-reachability relation. So we could replace Int/Int by a $Unit$ type in an annotation, and thus know that the division must be exact.

Moreover we can go somewhat further towards easing the generation of the annotations themselves, and indeed towards a limited type-inference. Global environments can be built up from declarations. The rules of Table 1 can be used to generate an annotation. We can generate the $\mathbf{E}'$ environment from the $\mathbf{E}$ environment by adding the productions derived from the labels, and adding type-bindings for local variables derived from declarations to $\mathbf{E}_i$. Type inference can be supported by using the requirement of exact division to introduce constraints on type-variables into the type-environment.

An experimental study of these problems is described in a separate report. The type-checking program described there is able to perform type-inference for local variables in a range of POP-11 programs, and has been demonstrated to work for some simple Forth programs as well.

References

[1] Barrett,R., Ramsay,A. and Sloman A., [1985] POP-11 A Practical Language for Artificial Intelligence, Ellis Horwood, Chichester, England and John Wiley N.Y.,USA.

[2] Burstall, R.M. and Popplestone, R.J., [1968] The POP-2 Reference Manual, Machine Intelligence 2, pp. 205-46, eds Dale,E. and Michie,D. Oliver and Boyd, Edinburgh, Scotland.

[3] Gordon, M.J.C., [1979] The Denotational Description of Programming Languages. p104. Springer Verlag, New York.

[4] Hopcroft and Ullman [1979], "An Introduction to Automata Theory, Languages and Computation", Addison-Wesley.