Towards Reasoning About Properties of Imperative Programs using Linear Logic

Daniel DaCosta
University of Minnesota
dacosta@cs.umn.edu

Abstract

In this paper we propose an approach to reasoning about properties of imperative programs. We assume in this context that the meanings of program constructs are described using rules in the natural semantics style with the additional observation that these rules may involve the treatment of state. Our approach involves modeling natural semantics style rules within a logic and then reasoning about the behavior of particular programs by reasoning about proofs in that logic. A key aspect of our proposal is to use a fragment of linear logic called Lolli (invented by Hodas and Miller) to model natural semantics style descriptions. Being based on linear logic, Lolli can provide logical expression to resources such as state. Lolli additionally possesses proof-theoretic properties that allow it to encode natural semantics style descriptions in such a way that proofs in Lolli mimic the structure of derivations based on the natural semantics rules. We will discuss these properties of Lolli and demonstrate how they can be exploited in modeling the semantics of imperative programs and in reasoning about such models.

1. Introduction

This paper concerns an approach to reasoning about the properties of imperative programs. Such programs, written in languages like Java and C, play an important role in safety- and security critical systems. They are pervasive, for example, in the software contained in medical devices and financial systems. Programs that malfunction in such contexts can lead to catastrophic system behavior. The underlying motivation for this work is that through the process of formal reasoning we can establish the absence of such bugs before these programs are run and thereby preclude undesirable behavior after their deployment.

Our objective in this work is not to reason about properties of particular programs but, rather, to develop a broad framework within which such reasoning may be conducted. An important ingredient of such a framework is a logic for describing the semantics of the programming language in which programs are constructed; a formalization of the semantics can then be combined with the description of a given program to model its overall behavior. An aspect that needs special treatment when dealing with imperative programs in this setting is the notion of state: imperative programs typically manipulate memory by storing and looking up values in relevant cells and how exactly they do this is important to understanding their behavior. Thus, the logic that we choose for our framework must facilitate the description as well as the analysis of the role of state in computations.

In constructing the framework we desire, we must also choose an approach to presenting the semantics of a programming language. We propose in this work to use the natural semantics style introduced by Kahn [12] for this purpose. Natural semantics style allows the meaning of a programming language construct to be modeled via derivations that closely reflect the actual computations that result from the construct. Thus, the process of reasoning about program behavior boils down naturally to reasoning about natural semantics style derivations. In our framework, programming language semantics will be modeled by translating these natural semantics descriptions into the underlying logic. This actually places two further constraints on the logic. First, it should have a structure that supports a natural encoding of natural semantics style descriptions. Second, the inference process in the logic should correspond transparently to the process of constructing natural semantics style derivations; this property allows reasoning about natural semantics style derivations to be reduced uniformly to reasoning about proofs in the logic.

The main thrust of the work in this paper is to identify a logic that satisfies the constraints described above and that would thereby be a suitable choice for encoding programs and programming language semantics within the framework we seek to design. We contend that linear logic, a logic of resources and actions invented by Jean-Yves Girard [7], provides a natural means for treating state-based aspects of computation and hence constitutes a good starting point. However, the logic we use needs also to allow for a treatment of natural semantics style descriptions. We argue that Lolli, a fragment of linear logic identified by Hodas and Miller [9], has such a character. To provide substance to our claim, we demonstrate how this logic can be used to model the semantics of a small collection of constructs in an imperative programming language. We further show how the meta-theoretic properties of Lolli allow us to translate an informal style of reasoning based on natural semantics derivations into reasoning about derivations in Lolli. Although a formalization of this reasoning process is beyond the scope of this work, we believe that this can be done following the approach used in the Abella system [5].

The rest of this paper is structured as follows. In the next section, we describe a simple imperative programming language and we present the meanings of the constructs in this language in natural semantics style. We then consider a small program in this language and show how we can organize the process of reasoning about it around natural semantics style derivations. This language and reasoning example provide us the means for explaining and defending our main contributions in the sections that follow. In Section 3 we present Lolli and we discuss the properties of derivations in it. In the following section we show how Lolli can be used to formalize the imperative programming language described earlier. In Section 5, we demonstrate how the properties of Lolli can be used in reasoning. In particular, we show that the informal reasoning process based on the natural semantics style presentation translates naturally into reasoning about derivations in Lolli. We conclude the paper in Section 6 with indications of directions in this work that may be worthwhile to pursue in the future.

2. A Simple Imperative Programming Language

In this section we define an imperative programming language and its evaluation semantics. We use these definitions to demonstrate how a program written in the language can be reasoned about informally.

2.1 The Language

The syntax of programs in our imperative programming language $\mathcal{L}$ is given by the following rules:

$$\begin{aligned} R ::= {} & \mathbb{N} \mid R + R \mid R - R \mid R > R \mid *R \\ & \mid R \leftarrow R \mid R; R \mid (R) \mid \mathit{while}\ R\ \mathit{do}\ R \end{aligned}$$

In this definition, the symbol $\mathbb{N}$ represents the category of expressions corresponding to the non-negative integers and the programs $*R, R_1 \leftarrow R_2$, and $R_1; R_2$ correspond to memory lookup (like in C), update of the value stored at R_2 to R_1, and evaluation of R_1 followed by evaluation of R_2, respectively. The last construct included by the syntax rules permits indefinite iteration in the language.

We will need a model of memory in order to present the semantics of the constructs in our language. Towards this end, we will represent memory as a partial function from natural numbers (denoted also in an overloaded fashion by $\mathbb{N}$) to expressions in the language that correspond to natural numbers. Given memory M, we will use the notation $M[x \mapsto y]$ to denote a modified memory given by the following partial function:

$$M[x \mapsto y](a) = \begin{cases} y & \text{if } x = a \\ M(a) & \text{if } x \neq a \end{cases}$$

Notice that although memory is modelled as a function from the conceptual domain of natural numbers, we will often want to do a “lookup” using the result of a computation. By an abuse of notation, we will allow memory to be “applied” to the expressions that denote natural numbers in the language.

We present the semantics of the constructs in our language by explaining what it means to evaluate them. We do this by defining an “evaluation relation” that we write as

$$\langle R, M \rangle \rightsquigarrow (N, M').$$

This relation is to be read as “the program R evaluated in memory M returns value N and modifies the memory to M'” or, when memory is not of interest, “R evaluates to value N.” When referring to components of this relation we may refer to the “input program expression”, the “input memory”, the “return value”, and the “output memory”, respectively. We define this relation through rules in the natural semantics style that are presented in Figure 1. For the uninitiated reader, each of these rules is to be read as asserting that the relation shown below the line holds if all the relations or properties above the line hold; the former is called the conclusion of the rule and the latter are called its premises. Notice that if a rule has no premises then its conclusion is unconditionally true.

A few comments are in order with regard to the rules in Figure 1. First, these rules are meant to be read as schemata: actual rules are to be generated by instantiating the schema variables N, N_1, N_2, and N_3 by (expressions denoting) numbers in $\mathbb{N}$, R, R_1, and R_2 by programs in $\mathcal{L}$, and M, M', M'', and M''' by memory. Second, in keeping with the systematic confusion of natural numbers with their representation in $\mathcal{L}$, we have also overloaded the operators $+$, $-$, $>$, $\leq$, and $\in$. Note also that with $-$ we associate the usual subtraction operation on natural numbers: $N_1 - N_2$ is 0 if N_1 is less than N_2. Finally, these rules make precise the interpretation that we would naturally think of associating with each of the constructs in the language. In this regard, the first five rules need no further explanation. The sixth and seventh rules encode the meaning of memory lookup and update, respectively: $*R$ causes R to be evaluated and the memory to be looked up at the resulting location while leaving the memory unchanged, whereas $R_1 \leftarrow R_2$ causes memory to be changed at the location corresponding to R_1 by the value corresponding to R_2. Notice also that the rightmost premise of the memory update rule ensures that the domain of memory remains fixed throughout evaluation. The

$$\dfrac{}{\langle N, M\rangle \rightsquigarrow (N, M)} \qquad \dfrac{\langle E_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle E_2, M'\rangle \rightsquigarrow (N_2, M'')}{\langle E_1{+}E_2, M\rangle \rightsquigarrow (N_1 + N_2, M'')} \qquad \dfrac{\langle E_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle E_2, M'\rangle \rightsquigarrow (N_2, M'')}{\langle E_1{-}E_2, M\rangle \rightsquigarrow (N_1 - N_2, M'')}$$

$$\dfrac{\langle E_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle E_2, M'\rangle \rightsquigarrow (N_2, M'') \qquad N_1 > N_2}{\langle E_1{>}E_2, M\rangle \rightsquigarrow (1, M'')} \qquad \dfrac{\langle E_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle E_2, M'\rangle \rightsquigarrow (N_2, M'') \qquad N_1 \leq N_2}{\langle E_1{>}E_2, M\rangle \rightsquigarrow (0, M'')}$$

$$\dfrac{\langle R, M\rangle \rightsquigarrow (N, M') \qquad M'(N) = N'}{\langle *R, M\rangle \rightsquigarrow (N', M')} \qquad \dfrac{\langle R_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle R_2, M'\rangle \rightsquigarrow (N_2, M'') \qquad M''(N_1) = N_3}{\langle R_1 \leftarrow R_2, M\rangle \rightsquigarrow (N_2, M''[N_1 \mapsto N_2])}$$

$$\dfrac{\langle R_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle R_2, M'\rangle \rightsquigarrow (N_2, M'')}{\langle R_1; R_2, M\rangle \rightsquigarrow (N_2, M'')}$$

$$\dfrac{\langle R_1, M\rangle \rightsquigarrow (0, M')}{\langle \mathit{while}\ R_1\ \mathit{do}\ R_2, M\rangle \rightsquigarrow (0, M')} \qquad \dfrac{\langle R_1, M\rangle \rightsquigarrow (N_1, M') \qquad \langle R_2, M'\rangle \rightsquigarrow (N_2, M'') \qquad \langle \mathit{while}\ R_1\ \mathit{do}\ R_2, M''\rangle \rightsquigarrow (N_3, M''') \qquad N_1 > 0}{\langle \mathit{while}\ R_1\ \mathit{do}\ R_2, M\rangle \rightsquigarrow (N_3, M''')}$$

Figure 1. Evaluation semantics for the imperative language $\mathcal{L}$

last three rules make precise the meaning of sequencing and of $while$ as an iteration construct.

When building derivations, we may build derivations for premises in any order provided the constrains between premises are met. However, we may significantly simplify the process of proof construction if we build them sequentially from the left most premise to the right most premise. Observe that adopting such a derivation building strategy does not limit the derivations that can be built.

2.2 Derivations as Computations

The rules defining the evaluation semantics provide us a means for constructing derivations of particular evaluation relations. Such derivations can be understood as an abstract view of the computation that results from particular programs. For example, suppose we are given a particular program R and a starting memory M and we desire to understand what value this program computes and what impact it has on memory. In this case, would pick two "meta variables" N and M' and we attempt to construct a derivation for the evaluation relation

$$\langle R, M\rangle \rightsquigarrow (N, M')$$

with the proviso that we may instantiate N and M' as needed along the way. Note also that the result of a computation must in fact be validated by success in constructing such a derivation. Thus, by analyzing all the possible derivations we also obtain a means for establishing properties of computations.

To illustrate the connection between derivations and computations in this setting, let us consider the program

$$2 \leftarrow *0; (0 \leftarrow *1; 1 \leftarrow *2)$$

and its evaluation in some memory M defined at locations $0, 1,$ and 2. This program swaps the values stored at two locations using a third location as temporary storage. We will build a derivation piecemeal, showing that for some N, the evaluation relation

$$\langle 2 \leftarrow *0; (0 \leftarrow *1; 1 \leftarrow *2), M\rangle \rightsquigarrow$$
$$(N, M[2 \mapsto M(0)][0 \mapsto M(1)][1 \mapsto M(0)])$$

holds.

Let X be the following derivation for $\langle 2 \leftarrow *0, M\rangle \rightsquigarrow (M(0), M')$ where $M' = M[2 \mapsto M(0)]$:

$$\dfrac{\dfrac{}{\langle 2, M\rangle \rightsquigarrow (2, M)} \qquad \dfrac{\dfrac{}{\langle 0, M\rangle \rightsquigarrow (0, M)}}{\langle 2 \leftarrow *0, M\rangle \rightsquigarrow (M(0), M)} \qquad \dfrac{}{M(2) = N_1}}{\langle 2 \leftarrow *0, M\rangle \rightsquigarrow (M(0), M')}$$

Let Ψ be the following derivation for $\langle 0 \leftarrow *1, M'\rangle \rightsquigarrow (M'(1), M'')$ where $M'' = M'[0 \mapsto M'(1)]$:

$$\dfrac{\dfrac{}{\langle 0, M'\rangle \rightsquigarrow (0, M')} \qquad \dfrac{\dfrac{}{\langle 1, M'\rangle \rightsquigarrow (1, M')}}{\langle *1, M'\rangle \rightsquigarrow (M'(1), M')} \qquad \dfrac{}{M'(0) = N_2}}{\langle 0 \leftarrow *1, M'\rangle \rightsquigarrow (M'(1), M'')}$$

Finally, let Ω be the following derivation for $\langle 1 \leftarrow *2, M''\rangle \rightsquigarrow (M''(2), M''')$ where $M''' = M''[1 \mapsto M''(2)]$:

$$\dfrac{\dfrac{}{\langle 1, M''\rangle \rightsquigarrow (1, M'')} \qquad \dfrac{\dfrac{}{\langle 2, M''\rangle \rightsquigarrow (2, M'')}}{\langle *2, M''\rangle \rightsquigarrow (M''(2), M'')} \qquad \dfrac{}{M''(1) = N_3}}{\langle 1 \leftarrow *2, M''\rangle \rightsquigarrow (M''(2), M''')}$$

Then we can combined X, Ψ and Ω to obtain the complete derivation that is shown below for the complete program expression of interest:

$$\dfrac{X \qquad \dfrac{\Psi \qquad \Omega}{\langle 0 \leftarrow *1; 1 \leftarrow *2, M'\rangle \rightsquigarrow (M''(2), M'')}}{\langle 2 \leftarrow *0; (0 \leftarrow *1; 1 \leftarrow *2), M\rangle \rightsquigarrow (M''(2), M''')}$$

To arrive at the desired conclusion, we have to show that M''', the memory at the end of the computation, is equivalent to

$$M[2 \mapsto M(0)][0 \mapsto M(1)][1 \mapsto M(0)].$$

Substituting the definition of M' in the definition of M'' yields

$$M[2 \mapsto M(0)][0 \mapsto M[2 \mapsto M(0)](1)].$$

By observing that

$$M[2 \mapsto M(0)](1) = M(1)$$

we have

$$M'' = M[2 \mapsto M(0)][0 \mapsto M(1)].$$

Replacing this result for M'' in the definition of M''' we get

$$\begin{array}{l} M[2 \mapsto M(0)][0 \mapsto M(1)][1 \mapsto \\ \qquad M[2 \mapsto M(0)][0 \mapsto M(1)](2)] \end{array}$$

Finally, by observing that

$$M[2 \mapsto M(0)][0 \mapsto M(1)](2) = M(0)$$

we arrive at the conclusion we want:

$$M''' = M[2 \mapsto M(0)][0 \mapsto M(1)][1 \mapsto M(0)].$$

2.3 Informal Reasoning about Imperative Programs

As we have explained earlier, we can extract information about the behavior of a program by analyzing the derivations that result from it. We illustrate this possibility in this subsection by showing how to demonstrate the correctness of a program for calculating the sum of the integers from 0 to a particular number N. Our argument at this stage will be informal; later sections will discuss a framework for formalizing this style of argument.

Let U be the following program:

$$\mathit{while}\ (*1){>}0\ \mathit{do}\ 0 \leftarrow *0{+}*1; 1 \leftarrow *1{-}1 \qquad (1)$$

Consider the program V written to calculate the value of $\sum_{i=0}^{N} i$ constructed with U:

$$0 \leftarrow 0; (1 \leftarrow N; U) \qquad (2)$$

We will show that given any N and any memory defined at 0 and 1, V calculates the correct answer and stores it in memory.

Lemma 1 (Total Correctness of U using structural operational semantics)**.** *$\forall N_1, N_2, M$ if $N_1, N_2 \in \mathbb{N}$ and M is memory where $M(0) = N_2$ and $M(1) = N_1$ then $\exists M'$ such that $\langle U, M\rangle \rightsquigarrow (0, M')$ and $M'(0) = N_2 + \sum_{i=0}^{N} i$*

Proof of Lemma 1 . This will be proven by induction on N_1. If $N_1 = 0$ then the following derivation can be constructed and $M'(0) = N_2 = N_2 + \sum_{i=0}^{0} i$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 1, M\rangle \rightsquigarrow (1, M)}}{\langle *1, M\rangle \rightsquigarrow (0, M)} \qquad \dfrac{}{\langle 0, M\rangle \rightsquigarrow (0, M)} \qquad \dfrac{}{0 \leq 0}}{\langle (*1){>}0, M\rangle \rightsquigarrow (0, M)}}{\langle \mathit{while}\ (*1){>}0\ \mathit{do}\ U, M\rangle \rightsquigarrow (N, M)}$$

If $M(1) = N_1$ and it is assumed this lemma holds for all memory W where $W(1) < N_1$ then the following derivation can be constructed:

$$\dfrac{X \qquad \dfrac{}{1 > 0} \qquad \Psi \qquad \begin{array}{c} \Omega \\ \langle \mathit{while}\ (*1){>}0\ \mathit{do}\ U, M''\rangle \rightsquigarrow (0, M''') \end{array}}{\langle \mathit{while}\ (*1){>}0\ \mathit{do}\ U, M\rangle \rightsquigarrow (0, M''')}$$

In this derivation, we let $M' = M[0 \mapsto M(0) + M(1)]$, $M'' = M'[1 \mapsto M(1) - 1]$, and M''' is the result memory from our inductive hypothesis. Let X be a derivation with an end-sequent of $\langle (*1){>}0, M\rangle \rightsquigarrow (1, M)$ and Ψ be a derivation with an end-sequent of $\langle 0 \leftarrow *0{+}*1; 1 \leftarrow *1{-}1, M\rangle \rightsquigarrow (M(1) - 1, M'')$. Both of these derivations can be constructed but are omitted; they are uninteresting with respect to this case. Since $M'(1) < N_1$ the inductive hypothesis can be used to give a derivation for Ω. □

From Lemma 1, the following theorem is easily shown:

Theorem 2 (Total Correctness of V using structural operational semantics)**.** *$\forall N, M$ if $N \in \mathbb{N}$ and M is memory defined at $M(0)$ and $M(1)$ then $\exists M'$ such that $\langle V, M\rangle \rightsquigarrow (0, M')$ and $M'(0) = \sum_{i=0}^{N} i$.*

Proof of Theorem 2. By case analysis on the derivation for $\langle V, M\rangle \rightsquigarrow (0, M')$ it suffices to show there is a derivation for $\langle U, (M[0 \mapsto 0])[1 \mapsto N]\rangle \rightsquigarrow (0, M')$ where $M'(0) = \sum_{i=0}^{N} i$. This is shown using Lemma 1.

□

3. The Specification Logic

In this section we present Lolli, the fragment of linear logic that we will use to formalize our imperative programming language. The first subsection introduces the language of Lolli and clarifies the meaning of its logical symbols through inference rules. This part of our presentation emphasizes the declarative nature of Lolli. When we use it to model natural semantics style descriptions, we would also like to be able to capture the structure of natural semantics style derivations. Towards this end, we show in the second subsection the relative completeness of goal-directed reasoning in Lolli. This discussion culminates in a reduced proof system for Lolli that we use exclusively in the rest of the paper.

3.1 The logic Lolli

Lolli is a logic that is built on the simply typed λ-calculus of Church [4]. The types underlying its language are constructed from a collection of primitive types that contain o, the type of propositions, and at least one other type; for the moment, we assume ι to be the only such type, but we will add to this collection as needed in later sections. The remaining types build on these primitive types using the function type constructor: if τ_1 and τ_2 are types, then $\tau_1 \rightarrow \tau_2$ is also a type and it denotes the collection of functions from τ_1 to τ_2.

The terms of Lolli are constructed from collections of typed variables and constants using the usual abstraction and application operations: the former yields the term $\lambda x.t$ of type $\tau_1 \rightarrow \tau_2$ given the term t of type τ_2 and the variable x of type τ_1, and the latter yields the term $(t_1\ t_2)$ of type τ_2 given terms t_1 and t_2 of types $\tau_1 \rightarrow \tau_2$ and τ_1 respectively. Abstraction is a binding operation that defines a scope for the variable, a concept that we will assume the reader to be familiar with. Two terms are considered to be equal if one can be obtained from the other by some sequence of α-conversions, i.e. the replacement of a subpart of the form $\lambda x.t$ by $\lambda y.t'$ provided x and y are variables of the same type, y does not appear free in t and t' results from t by the replacement of the free occurrences of x by y. Given a term s of the same type as x, we will write $t[s/x]$ to denote the result of substituting s for the free occurrences of x in t in a capture avoiding way; notice that in correctly carrying out such a substitution, we may need to apply some α-conversions. A term t is said to be obtained by β-contraction from another term s if it results from replacing a subterm of s that has the form $((\lambda x.t_1)\ t_2)$ by $t_2[t_1/x]$. Two terms are also considered equal if one can be obtained from the other by some sequence of applications of β-contractions or its inverse. We will use this notion of equality implicitly in the rest of this paper. In the context of the simply typed λ-calculus, it is known that every term has a normal form modulo β-contractions, i.e. it is equal to a term which does not contain a subterm of the form $((\lambda x.t_1)\ t_2)$. We will depict terms solely by their normal forms.

Lolli has a set of constants that serve to build a logic over its terms. These constants, referred to as *logical constants* consist of the following: $\&$, $\multimap$, $\Rightarrow$, $\otimes$, and $\oplus$ all of type $o \rightarrow (o \rightarrow o)$ and written in infix form; ! of type $o \rightarrow o$; and, for each type τ, the constants $\forall_\tau$ and $\exists_\tau$ with type $(\tau \rightarrow o) \rightarrow o$. The constants $\forall_\tau$ and $\exists_\tau$ are referred to as quantifiers and the remaining constants constitute the logical connectives. In addition to these constants, expressions in Lolli may also be formed from user defined constants, referred to as *nonlogical constants*. The well-formed terms of type o in Lolli are distinguished as *formulas*. Notice that a formula may have as its top-level symbol a logical constant, a variable or a nonlogical constant. In the latter two cases, the formula is said to be *atomic*. Further, it is a rigid atom if its top-level symbol is a nonlogical constant. We shall use the syntactic variable A to denote atomic formulas and A_r to denote rigid atoms.

At a logical level, Lolli is oriented towards proving judgments represented by *sequents*. Formally, a sequent is an object of the form

$$\Gamma; \Delta \vdash G$$

where Γ is a set of formulas, Δ is a multiset of formulas and G is a formula. Intuitively, such a sequent corresponds to the claim that G, the *goal formula*, is derivable given the resources Γ and Δ. The resources in Γ are distinguished as being *unbounded*: formulas in Γ would typically be used to represent unchanging facts in a specification setting, such as the natural semantics rules governing the behavior of imperative programs. On the other hand, formulas in Δ constitute *bounded* resources: referring again to the imperative programming example, they may be used to represent the state of memory at a particular point in computation.

The syntax of formulas that may be used as resources and goals is limited in Lolli. Specifically, they may only be the P and G formulas described by the syntax rules below:

$$\begin{aligned} P &::= \ A_r \mid P \,\&\, P \mid G \multimap P \mid G \Rightarrow P \mid \forall x.P \\ G &::= \ \top \mid A \mid G \,\&\, G \mid P \multimap G \mid P \Rightarrow G \mid \forall x.G \\ &\qquad \mid \exists x.G \mid !G \mid G \otimes G \mid G \oplus G \end{aligned} \tag{3}$$

We refer to P formulas also as *program clause formulas*. Notice that the connectives $\&$, $\multimap$, $\Rightarrow$, and $\forall$ are allowed in both kinds of formulas. However, there are differing constraints in the use of $\multimap$ and $\Rightarrow$. When these are used in the resource formulas, the formula on the left must be a goal formula and that on the right must be a resource formula. When they are used in a goal formula on the other hand, the formula on the left must be a resource formula and that on the right must be a goal formula. As we shall see presently, these restrictions play an important role in maintaining the structure of sequents in the course of a derivation and therefore in the coherence of the inference rules for Lolli. In addition to the already mentioned connectives, goal formulas may contain $\top$, $\exists$, !, $\otimes$, and $\oplus$.

The rules for deriving sequents in Lolli are presented in Figure 2. The sequent that appears below the line in each of these rules is called its conclusion and the sequents that appear above the line constitute its premises. The L or R in the labels of these rules denotes whether the rule introduces a logical symbol on the left or the right of the $\vdash$. Grouped by L or R they may be referred to as left-introduction rules and right-introduction rules, respectively. In the rules pertaining to the logical symbols, the formula in the conclusion that contains the introduced symbol is called the *principal formula*. This terminology is extended to the id rule and $absorb$ rules to denote the formulas represented by A and B, respectively. When we write Γ, F in the unbounded context in these rules, we mean it to denote $\Gamma \cup \{F\}$, i.e. F may also be contained in Γ. On the other hand, in the unbounded context Δ, F represents $\Delta \uplus \{F\}$, i.e. Δ constitutes the bounded resources with the exclusion of the selected copy of the formula F. Relatedly, Δ_1, Δ_2 in such a context stands for $\Delta_1 \uplus \Delta_2$, i.e., the comma represents multiset union.

$$\dfrac{}{\Gamma; A \vdash A}\,id \qquad \dfrac{\Gamma, B; \Delta, B \vdash G}{\Gamma, B; \Delta \vdash G}\,absorb \qquad \dfrac{}{\Gamma; \Delta \vdash \top}\,\top R \qquad \dfrac{\Gamma; \Delta, B_i \vdash G}{\Gamma; \Delta, B_1 \& B_2 \vdash G}\,\& L(i \in \{1,2\}) \qquad \dfrac{\Gamma; \Delta \vdash G_1 \qquad \Gamma; \Delta \vdash G_2}{\Gamma; \Delta \vdash G_1 \& G_2}\,\& R$$

$$\dfrac{\Gamma; \Delta_1 \vdash B_1 \qquad \Gamma; \Delta_2, B_2 \vdash G}{\Gamma; \Delta_1, \Delta_2, B_1 \multimap B_2 \vdash G} \multimap L \qquad \dfrac{\Gamma; \Delta, G_1 \vdash G_2}{\Gamma; \Delta \vdash G_1 \multimap G_2} \multimap R \qquad \dfrac{\Gamma; \emptyset \vdash B_1 \qquad \Gamma; \Delta, B_2 \vdash G}{\Gamma; \Delta, B_1 \Rightarrow B_2 \vdash G} \Rightarrow L \qquad \dfrac{\Gamma, G_1; \Delta \vdash G_2}{\Gamma; \Delta \vdash G_1 \Rightarrow G_2} \Rightarrow R$$

$$\dfrac{\Gamma; \Delta, (B\ t) \vdash G}{\Gamma; \Delta, \forall x.B \vdash G}\,\forall L \qquad \dfrac{\Gamma; \Delta \vdash Gc}{\Gamma; \Delta \vdash \forall x.G}\,\forall R \qquad \dfrac{\Gamma; \Delta \vdash (G\ t)}{\Gamma; \Delta \vdash \exists x.G}\,\exists R$$

$$\dfrac{\Gamma; \emptyset \vdash G}{\Gamma; \emptyset \vdash !G}\,!R \qquad \dfrac{\Gamma; \Delta \vdash G_i}{\Gamma; \Delta \vdash G_1 \oplus G_2}\,\oplus R(i \in \{1,2\}) \qquad \dfrac{\Gamma; \Delta_1 \vdash G_1 \qquad \Gamma; \Delta_2 \vdash G_2}{\Gamma; \Delta_1, \Delta_2 \vdash G_1 \otimes G_2}\,\otimes R$$

Figure 2. The inference rules in Lolli. In the $\forall R$ rule, c must not occur in Γ, Δ, or G. In the $\forall L$ and $\exists R$ rules, the term t generalized upon must be such that $(B\ t)$ and $(G\ t)$ are a program clause formula and a goal formula, respectively.

Some comments on the inference rules are useful both in understanding the logical structure of Lolli and the intended meaning of the logical symbols. The rules for the use of resource formulas in Lolli are all stated with respect to the bounded context. The only exception to this is the $absorb$ rule which encodes the possibility of making a copy of an unbounded resource before using it in a bounded fashion. The rules for the quantifiers give them their usual interpretation with the caveat that the domain of quantification is restricted so as to preserve the normal form of sequents in Lolli. The formula $G_1 \otimes G_2$ is interpreted as saying that there are enough resources to show both G_1 and G_2: the rule for proving this formula requires each component to be shown from a partitioning of the bounded resources. The connectives $\&$ and $\oplus$ are meant to encode different kinds of choices. The formula $G_1 \ \&\ G_2$ signifies that the available resources are sufficient to satisfy either G_1 or G_2, whichever one we choose. Accordingly, to prove a sequent that has such a formula on the right of $\vdash$, we have to show that we can prove sequents with the same resources and each of G_1 and G_2 as the goal. On the other hand, if the formula $B_1 \ \&\ B_2$ is available as a resource, this means that we can choose which one of the components we actually want to use, something that underlies the left-introduction rule for this connective. In contrast, the formula $G_1 \oplus G_2$ means that we can have one of G_1 or G_2 based on the resources, but we do not know *a priori* which. Correspondingly, to prove a sequent that has such a formula on the right of $\vdash$, it suffices to prove a sequent with the same resources and with one of G_1 or G_2 as the goal. The $\multimap$ connective captures a notion of resource conversion: To show $G_1 \multimap G_2$ we must somehow use G_1 in showing G_2 and, conversely, when given $B_1 \multimap B_2$, we may consume some of the resources to show B_1 and then use B_2 itself as a resource. The $\Rightarrow$ connective also represents resource conversion, but this time an unbounded resource. Note that the rules for $\multimap$ and $\Rightarrow$ may move formulas from one side of $\vdash$ to the other and could potentially result in destroying the form of permitted sequents in Lolli. However, the restriction on what can appear on either side of $\multimap$ and $\Rightarrow$ in goal and program clause formulas ensures that this does not happen. The ! connective corresponds to treating its argument as being independent of the finite resources. The id rule cements the fact that all the bounded resources must be consumed in a derivation. In this setting $\top$ corresponds to a "sink" or a garbage collector for the bounded resources.

We illustrate the rules of Lolli by considering a few proofs that use them. First, consider the sequent

$$\emptyset; \emptyset \vdash (A_1 \ \&\ A_2) \Rightarrow (A_1 \otimes A_2).$$

This sequent expresses the intuition that if we have $A_1 \ \&\ A_2$ as an unbounded resource, then we must simultaneously have both A_1 and A_2 provided our bounded resources are empty. A derivation for the sequent is shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \ \&\ A_2; A_2 \vdash A_2}\,id \qquad \dfrac{}{A_1 \ \&\ A_2; A_1 \vdash A_1}\,id}{A_1 \ \&\ A_2; A_1, A_2 \vdash A_1 \otimes A_2}\,\otimes R}{A_1 \ \&\ A_2; A_1, A_1 \ \&\ A_2 \vdash A_1 \otimes A_2}\,\& L}{A_1 \ \&\ A_2; A_1 \ \&\ A_2, A_1 \ \&\ A_2 \vdash A_1 \otimes A_2}\,\& L}{A_1 \ \&\ A_2; A_1 \ \&\ A_2 \vdash A_1 \otimes A_2}\,absorb}{A_1 \ \&\ A_2; \emptyset \vdash A_1 \otimes A_2}\,absorb}{\emptyset; \emptyset \vdash (A_1 \ \&\ A_2) \Rightarrow (A_1 \otimes A_2)} \Rightarrow R$$

This proof uses the $\& L$ and $absorb$ rules in a situation when the formula on the right $\vdash$ is $A_1 \otimes A_2$, i.e., is not atomic. Such a proof is not goal-directed, i.e., if we think of the process of searching for a proof for the given sequent, the formula on the right of the $\vdash$ symbol does not guide the choice of rule to use to arrive at the conclusion. In the next subsection we will consider the idea of uniform proofs that will provide us a means for restricting attention to only goal-directed proofs.

Notice that the unbounded availability of $A_1 \ \&\ A_2$ is important to the above proof: if we change the sequent to

$$\emptyset; \emptyset \vdash (A_1 \ \&\ A_2) \multimap (A_1 \otimes A_2)$$

then it is no longer provable. Given the formula $A_1 \,\&\, A_2$ as a bounded resource, we have to make a choice between using with A_1 or A_2, also as a bounded resource. This choice is mutually exclusive; we may not have both A_1 and A_2. On the other hand, if $A_1 \otimes A_2$ is on the right of $\vdash$, then both A_1 and A_2 must be available as (bounded) resources for the sequent to be provable.

The process of finding proofs for sequents typically involves search. Two common strategies that are used in this setting are *forward chaining* and *backward chaining*. These strategies refer to how we use implicational formulas, which in Lolli could be ones that have either $\multimap$ or $\Rightarrow$ as their top-level connective, available in resources in guiding the search. In the former case, we use the fact that the lefthand side of the implication is already available as a resource and we then reason forward, by adding the righthand side as a resource. In the latter case, we observe that the goal formula of the sequent matches the righthand side of the implication and then reduce the task to showing the lefthand side from the available resources. The following proof can be understood as the result of using a forward chaining strategy to prove the sequent $A_1; A_1 \multimap A_2, A_2 \multimap A_3 \vdash A_3$.

$$\dfrac{\dfrac{\dfrac{}{A_1; A_1 \vdash A_1}\,id}{A_1; \emptyset \vdash A_1}\,absorb \qquad \dfrac{\dfrac{}{A_1; A_2 \vdash A_2}\,id \qquad \dfrac{}{A_1; A_3 \vdash A_3}\,id}{A_1; A_2, A_2 \multimap A_3 \vdash A_3}\multimap L}{A_1; A_1 \multimap A_2, A_2 \multimap A_3 \vdash A_3}\multimap L$$

In Section 3.2, we will consider a different proof resulting from a backward chaining strategy for this sequent.

3.2 A Reduced Proof System for Lolli

The derivation system for Lolli that we saw in the previous subsection presents us with alternative ways to construct a proof. For example, we may have the choice of using either a left or a right rule at a particular point in proof. In modelling natural semantics style rules for imperative programming languages, we will want to use sequents in a specific way: the unbounded context will encode the semantics of programming constructs, the bounded context will model the state and the goal formula will represent the program producing the computation. If we are to analyze the properties of programs using this setup, it would be ideal if we could focus our attention on Lolli proofs that closely follow program behavior. We show here that this is possible. In particular, we demonstrate that, from a provability perspective, it suffices to look at proofs that are goal-directed in that, when looking at derivations bottom up, the first step is always to simplify a complex goal formula.

The following definition, first introduced by Miller*et al*[16], provides an encapsulation of the idea of goal-directedness in the context of Lolli proofs. It was or

Definition 3 (Uniform Proof)**.** A uniform proof is a Lolli proof in which every sequent with a non-atomic goal formula on the right of $\vdash$ is the conclusion of an inference rule that introduces the top-level logical symbol of that formula.

Towards understanding uniform provability, consider the proof shown in Section 3.1 for the sequent

$$\emptyset; \emptyset \vdash (A_1 \,\&\, A_2) \Rightarrow (A_1 \otimes A_2).$$

That proof is not a uniform proof. In that proof, there are two $absorb$ rules and two $\&L$ rules that have as a conclusion a sequent in which the goal formula $A_1 \otimes A_2$ appears as the right of $\vdash$. However, the same sequent does have a uniform proof that is shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_1 \,\&\, A_2; A_1 \vdash A_1}\,id}{A_1 \,\&\, A_2; A_1 \,\&\, A_2 \vdash A_1}\,\&L}{A_1 \,\&\, A_2; \emptyset \vdash A_1}\,absorb \qquad \dfrac{\dfrac{\dfrac{}{A_1 \,\&\, A_2; A_2 \vdash A_2}\,id}{A_1 \,\&\, A_2; A_1 \,\&\, A_2 \vdash A_2}\,\&L}{A_1 \,\&\, A_2; \emptyset \vdash A_2}\,absorb}{A_1 \,\&\, A_2; \emptyset \vdash A_1 \otimes A_2}\otimes R}{\emptyset; \emptyset \vdash (A_1 \,\&\, A_2) \Rightarrow (A_1 \otimes A_2)}\Rightarrow R$$

In fact, every provable Lolli sequent has a uniform proof as we now show.

Theorem 4 (Lolli Admits Uniform Provability)**.** *The sequent* $\Gamma; \Delta \vdash G$ *has a proof in Lolli if and only if it has a uniform proof.*

Proof. The "if" direction is obvious. For the "only if" direction, we consider a proof that is not uniform and show how to transform it into a uniform proof. We associate with a proof a non-uniformity measure that counts the number of inference rule occurrences that do not act on a complex goal formula that appears to the right of $\vdash$ in their conclusion. If this measure is non-zero, we show how to reduce it by 1. The conclusion then follows by induction on the measure.

If a proof has a non-zero non-uniformity measure, then there must be a path in it in which there is a first occurrence of a left rule that has a complex goal formula to the right of $\vdash$ in its conclusion. We show how to reduce the height of this path by 1. By induction on this height it follows that we can eliminate this violation of uniformity and thereby reduce the non-uniformity measure of the proof. Observe that since the rule in question is the first one along the path to violate the uniformity property, it must be preceded in the proof by a right rule. We use this fact in our argument. In particular, we consider the possible cases for the right and left rules and show that the left rule can be permuted above the right one, thereby moving the violation of non-uniformity closer to a leaf.

In a detailed consideration of the cases, it is useful to categorize rules based on the number of premises they have. Category I will represent rules with one premise and category II will represent rules with two premises.

Suppose that the case in question involves two inference rules from category I. An example of such a situation is the following:

$$\dfrac{\dfrac{\dfrac{\Xi}{\Gamma; \Delta, B_i, G_1 \vdash G_2}}{\Gamma; \Delta, B_i \vdash G_1 \multimap G_2}\multimap R}{\Gamma; \Delta, B_1 \,\&\, B_2 \vdash G_1 \multimap G_2}\,\&L(i \in \{1, 2\})$$

This proof can be rearranged as follows:

$$\dfrac{\dfrac{\overset{\Xi}{\Gamma;\Delta,B_i,G_1\vdash G_2}}{\Gamma;\Delta,B_1\mathbin{\&}B_2,G_1\vdash G_2}\,\&L(i\in\{1,2\})}{\Gamma;\Delta,B_1\mathbin{\&}B_2\vdash G_1\multimap G_2}\multimap R$$

By permuting the left rule above the right one, we have reduced the length of the path by 1 as required. A similar argument applies to all the other cases of rules in these two respective categories.

Suppose that the case in question involves a right inference rule from category I and a left inference rule from category II. An example of this kind is presented by the following derivation:

$$\dfrac{\overset{\Psi}{\Gamma;\Delta_1\vdash B_1}\qquad\dfrac{\overset{\Xi}{\Gamma;\Delta_2,B_2\vdash G_i}}{\Gamma;\Delta_2,B_2\vdash G_1\oplus G_2}\oplus R(i\in\{1,2\})}{\Gamma;\Delta_1,\Delta_2,B_1\multimap B_2\vdash G_1\oplus G_2}\multimap L$$

In this case the derivation can be rearranged as follows to once again reduce the length of the path by 1:

$$\dfrac{\dfrac{\overset{\Psi}{\Gamma;\Delta_1\vdash B_1}\qquad\overset{\Xi}{\Gamma;\Delta_2,B_2\vdash G_i}}{\Gamma;\Delta_1,\Delta_2,B_1\multimap B_2,\vdash G_i}\multimap L}{\Gamma;\Delta_1,\Delta_2,B_1\multimap B_2\vdash G_1\oplus G_2}\oplus R(i\in\{1,2\})$$

The other cases for rules from the categories under consideration are similar.

Suppose the case in question involves a right inference rule from category II and a left inference rule instance from category I. An example of this kind is the following:

$$\dfrac{\dfrac{\overset{\Psi}{\Gamma;\Delta_1,B_i\vdash G_1}\qquad\overset{\Xi}{\Gamma;\Delta_2\vdash G_2}}{\Gamma;\Delta_1,\Delta_2,B_i\vdash G_1\otimes G_2}\otimes R}{\Gamma;\Delta_1,\Delta_2,B_1\mathbin{\&}B_2\vdash G_1\otimes G_2}\,\&L(i\in\{1,2\})$$

Here again, we can permute the left inference rule above the right one as follows:

$$\dfrac{\dfrac{\overset{\Psi}{\Gamma;\Delta_1,B_i\vdash G_1}}{\Gamma;\Delta_1,\Delta_2,B_1\mathbin{\&}B_2\vdash G_1\otimes G_2}\,\&L\qquad\overset{\Xi}{\Gamma;\Delta_2\vdash G_2}}{\Gamma;\Delta_1,\Delta_2,B_1\mathbin{\&}B_2\vdash G_1\otimes G_2}\otimes R$$

The other cases under this combination are treated similarly.

Finally, suppose that the situation under consideration involves a right and left inference rule both from category II. An example of this kind is the following:

$$\dfrac{\overset{\Psi}{\Gamma;\Delta_1\vdash B_1}\qquad\dfrac{\overset{\Xi}{\Gamma;\Delta_2,B_2\vdash G_1}\qquad\overset{\Theta}{\Gamma;\Delta_3\vdash G_2}}{\Gamma;\Delta_2,\Delta_3,B_2\vdash G_1\otimes G_2}\otimes R}{\Gamma;\Delta_1,\Delta_2,\Delta_3,B_1\multimap B_2\vdash G_1\otimes G_2}\multimap L$$

Here we rearrange the derivation as follows, again obviously reducing the length of the path to the errant left rule by one.

$$\dfrac{\dfrac{\overset{\Psi}{\Gamma;\Delta_1\vdash B_1}\qquad\overset{\Xi}{\Gamma;\Delta_2,B_2\vdash G_1}}{\Gamma;\Delta_1,\Delta_2,B_1\multimap B_2\vdash G_1}\multimap L\qquad\overset{\Theta}{\Gamma;\Delta_3\vdash G_2}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,B_1\multimap B_2\vdash G_1\otimes G_2}\otimes L$$

The other cases for the rules in the category under consideration are treated similarly. □

We are thinking of modeling natural semantics style inference rules using the $\multimap$ connective: modeled natural semantics rule conclusion relations will occur to the right, also known as the *head*, of a $\multimap$ and modeled premise relations will occur to left, also known as the *body*, of a $\multimap$. When modeled this way, $\multimap L$ application on formulas with heads matching atomic goals mimics natural semantics style derivation construction. A backward chaining proof search strategy is one where this process is repeated for proof construction.

The following definition captures the structure of proofs built using a backward chaining proof search strategy.

Definition 5 (Simple Proof)**.** A uniform proof is simple if every left introduction inference rule instance acts on a marked formula. A unique formula in the bounded context is marked if it is the principal formula of an id instance or if:

- P_1 or P_2 are marked in the premises sequent of a $\&L$ instance then the formula $P_1 \mathbin{\&} P_2$ is marked in the conclusion sequent.
- $P[t/x]$ is marked in the premise sequent of a $\forall L$ instance then the formula $\forall x.P$ is marked in the conclusion sequent.
- P is marked in the right-hand premise sequent of a $\multimap L$ instance then formula $G \multimap P$ is marked in the conclusion sequent.
- P is marked in the right-hand premise sequent of a $\Rightarrow L$ instance then formula $G \Rightarrow P$ is marked in the conclusion sequent.

The second proof from Section 3.1 is an example of a proof that is not simple. This can be illustrated by attempting to mark the proof according Definition 5. In the following proof, the dots indicate formulas which can be marked.

$$\dfrac{\dfrac{\dfrac{}{A_1;\dot{A}_1\vdash A_1}\,id}{A_1;\emptyset\vdash A_1}\,absorb\qquad\dfrac{\dfrac{}{A_1;\dot{A}_2\vdash A_2}\,id\qquad\dfrac{}{A_1;\dot{A}_3\vdash A_3}\,id}{A_1;A_2,A_2\overset{.}{\multimap}A_3\vdash A_3}\multimap L}{A_1;A_1\multimap A_2,A_2\multimap A_3\vdash A_3}\multimap L$$

Consider the bottom most $\multimap L$ instance principal formula $A_1 \multimap A_2$, call this instance one. According to the marking strategy, for this formula to be marked A_2 must be marked at the root of the right-hand premise sub-proof of instance one. Consequently, instance one acts on a unmarked formula.

The following proof is a simple proof for the same sequent. Observe that every principal formula of a left introduction rule is marked.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A_1;\dot{A}_1 \vdash A_1}\,id}{A_1;\emptyset \vdash A_1}\,Absorb \qquad \dfrac{}{A_1;\dot{A}_2 \vdash A_2}\,id}{A_1;A_1 \mathrel{\dot{\multimap}} A_2 \vdash A_2}\multimap L \qquad \dfrac{}{A_1;\dot{A}_3 \vdash A_3}\,id}{A_1;A_1 \multimap A_2, A_2 \mathrel{\dot{\multimap}} A_3 \vdash A_3}\multimap L$$

We now show that every provable sequent in Lolli has a simple proof.

Theorem 6 (The Original Specification Logic Admits Simple Provability)**.** *The sequent* $\Gamma;\Delta \vdash G$ *has a uniform proof in Lolli if and only if it has a simple proof in Lolli.*

Proof. This proof is similar to the proof given in Theorem 4. The "if" direction is obvious and in the "only if" direction, we associate with a proof a non-simple measure that counts the number of unmarked principal formulas occurring to the left of a $\vdash$. If this measure is non-zero, we show how to reduce it by 1. The conclusion then follows by induction on the measure.

Observe that if a non-simple instance occurs below an $absorb$ instance a permutation is immediate. Therefore, we restrict analysis to non-simple instances below instances that are not $absorb$.

Suppose the non-simple instance is a $\&$ or $\forall$ left introduction instance. Observe, that the rule above this non-simple instance must be a left introduction instance; A is atomic so no right introduction rules apply. If the rule is below a $\forall$ or $\&$ left introduction instance permutation of these instances is immediate. If the rule above is a left introduction instance of $\multimap$ a straightforward permutation is possible. We consider one such case in detail where Ψ and Ξ are simple proofs.

$$\dfrac{\dfrac{\begin{matrix}\Psi\\ \Gamma;\Delta_1,P_i \vdash B_1\end{matrix} \qquad \begin{matrix}\Xi\\ \Gamma;\Delta_2,P_2 \vdash A\end{matrix}}{\Gamma;\Delta_1,\Delta_2,P_1 \multimap P_2, P_i \vdash A}\multimap L}{\Gamma;\Delta_1,\Delta_2,P_1 \multimap P_2, P_3 \,\&\, P_4 \vdash A}\&L(i \in \{3,4\})$$

This non-simple uniform proof may be permuted to one with the following form:

$$\dfrac{\dfrac{\begin{matrix}\Psi\\ \Gamma;\Delta_1,P_i \vdash P_1\end{matrix}}{\Gamma;\Delta_1,P_3 \,\&\, P_4 \vdash P_1}\&L \qquad \begin{matrix}\Xi\\ \Gamma;\Delta_2,P_2 \vdash A\end{matrix}}{\Gamma;\Delta_1,\Delta_2,P_1 \multimap P_2, P_3 \,\&\, P_4 \vdash A}\multimap L$$

When the non-simple instance is a $\forall$ left introduction instance or the instance above is a $\Rightarrow$ the permutations differ only slightly.

Suppose the non-simple instance is a $\multimap$ or $\Rightarrow$ left introduction instance. Observe, that the right premise must begin with a left introduction instance; A is atomic so no right introduction rules apply. Furthermore, observe that the left premise is irrelevant with respect to marking. We consider one case in detail where Ψ and Ξ are simple proofs.

$$\dfrac{\begin{matrix}X\\ \Gamma;\Delta_3 \vdash P_3\end{matrix} \qquad \dfrac{\begin{matrix}\Psi\\ \Gamma;\Delta_1,P_4 \vdash P_1\end{matrix} \qquad \begin{matrix}\Xi\\ \Gamma;\Delta_2,P_2 \vdash A\end{matrix}}{\Gamma;\Delta_1,\Delta_2,P_1 \multimap P_2, P_4 \vdash A}\multimap L}{\Gamma;\Delta_1,\Delta_2,\Delta_3,P_1 \multimap P_2, P_3 \multimap P_4 \vdash A}\multimap L$$

This non-simple uniform proof may be permuted to one with the following form:

$$\dfrac{\dfrac{\begin{matrix}X\\ \Gamma;\Delta_3 \vdash P_3\end{matrix} \qquad \begin{matrix}\Psi\\ \Gamma;\Delta_1,P_4 \vdash P_1\end{matrix}}{\Gamma;\Delta_1,\Delta_3,P_3 \multimap P_4 \vdash P_1}\multimap L \qquad \begin{matrix}\Xi\\ \Gamma;\Delta_2,P_2 \vdash A\end{matrix}}{\Gamma;\Delta_1,\Delta_2,\Delta_3,P_1 \multimap P_2, P_3 \multimap P_4 \vdash A}\multimap L$$

The remaining permutations involving non-simple $\multimap$ and $\Rightarrow$ left introduction instances and follow this permutation closely. □

Instances of $absorb$ may appear anywhere prior to the use of its principal formula. Without further meta-theoretical results, natural semantics style derivation mimicry in Lolli will be modulo $absorb$ instance placement. The following definition prescribes an exact placement for all $absorb$ instance.

Definition 7 (Coincided Proof)**.** A coincided proof is a simple proof where every $absorb$ rule instance unbounded premise formula corresponding to the principal formula is the principal formula of a left introduction or identity rule instance directly above it.

The first proof in this section is not a coincided one because an $absorb$ instance is detached where the proof in Subsection 3.2 is a coincided one because all $absorb$ instances satisfy Definition 7.

Theorem 8 (Lolli Admits Conincided Provability)**.** *The sequent* $\Gamma;\Delta \vdash G$ *has a simple proof in Lolli if and only if it has a conincided proof in Lolli.*

Proof. This proof is similar to the previous ones. Observe that all coincided proofs are simple ones, this satisfies the "if" direction. Now consider the "only if" direction. It is easy to see that $absorb$ instances may be permuted up until they coincide with a left introduction or identity rule instance. From this, we may conclude the argument by induction on the measure of non-coincided $absorb$ rule instances. □

Theorems 4, 6, and 8 can be used to yield a reduced proof system that admits only coincided proofs. To do so we first inductively define a unary predicate $||P||$ where P is a program clause formula that captures a backward chaining proof search strategy. The predicate takes a program clause formula as an argument and returns a set of triples where the first, second, and third projection is a set of goal formulas, a multiset of goal formulas, and a program clause formula, respectively. Each triple represents unbounded(the first projection) and bounded(the second projection) proof obligations for some program clause formula. An unbounded proof obligation is one that must be provided strictly from the unbounded context and a bounded proof obligation is one that must be proved from some portion of the bounded context. Let $||P||$ be the smallest set such that:

1. $\langle \emptyset, \emptyset, A \rangle \in ||A||$

2. if $\langle \Gamma, \Delta, P_1 \,\&\, P_2 \rangle \in ||P||$ then both $\langle \Gamma, \Delta, P_1 \rangle \in ||P||$ and $\langle \Gamma, \Delta, P_2 \rangle \in ||P||$
3. if $\langle \Gamma, \Delta, \forall x.P \rangle \in ||P||$ then, for all closed terms t, $\langle \Gamma, \Delta, P[t/x] \rangle \in ||P||$
4. if $\langle \Gamma, \Delta, P_1 \Rightarrow P_2 \rangle \in ||P||$ then $\langle \Gamma \cup P_1, \Delta, P_2 \rangle \in ||P||$
5. if $\langle \Gamma, \Delta, P_1 \multimap P_2 \rangle \in ||P||$ then $\langle \Gamma, \Delta \uplus P_1, P_2 \rangle \in ||P||$

Let our specification logic have all right introduction rules from Figure 2 and the back chaining rules given in Figure 3.

There are two forms of backward chaining in this figure both having as their principal formula B. Intuitively, an instance of both could replace a series of left introduction instances in a coincided proof. Using the former requires that the left introduction series begin (in a bottom-up reading) with an $absorb$ instance.

Theorem 9 (The Specification Logic and Lolli Equivalence)**.** *The sequent $\Gamma; \Delta \vdash G$ has a proof in Lolli if and only if it has a proof in the specification logic.*

Proof. In the "if" direction, due to the definition of the backward chaining rules, any back chaining instance in the specification logic proof can be replace by some sequence of left introduction and $absorb$ instances from Lolli.

Now, consider the "only if" direction. Application of Theorems 4 followed by 6 and finally 8 allows us to convert a Lolli proof to a coincided proof. Finally, by Definition 5 and the definition of $||P||$, we may replace runs of left-introduction and $absorb$ instances by one of the two instances of backward chaining. □

4. Modeling Imperative Programming Languages

In this section, the imperative programming language defined in Section 2 is modeled using the specification logic presented in Section 3. Additionally, proof mimicry of derivations is demonstrated by considering the proof of a modeled evaluation relation and that evaluation relations derivation. Throughout this section and the rest of this paper, we refer to the imperative programming language and its evaluation semantics as the "object system".

4.1 The Model

Our model extends the kinds of types we may have. Types in our model will now include a type for programs in the model, $\mathbb{R}$ and for syntax representing natural numbers, $\mathbb{N}$ (again, overloaded for use in the specification logic).

Let $t_{\mathbb{R}}$ be a function that translates programs from $\mathcal{L}$ given in Section 2 into terms of type $\mathbb{R}$ in the specification logic.

$$t_{\mathbb{R}}(i) = \begin{cases} i & \text{if } i \in \mathbb{N} \\ (add\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = j{+}k \\ (sub\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = j{-}k \\ (gt\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = j{>}k \\ (get\ t_{\mathbb{R}}(j)) & \text{if } i = {*}j \\ (set\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = j \leftarrow k \\ (seq\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = j; k \\ (wh\ t_{\mathbb{R}}(j)\ t_{\mathbb{R}}(k)) & \text{if } i = \text{while } j \text{ do } k \end{cases} \tag{4}$$

Observe that any function can be represented as set of tuples relating inputs to outputs. Such representations are often referred to as function graphs. Let t_m be a recursive function that translates memory function graphs to multisets composed exclusively of occurrences of the binary predicate m with the type $\mathbb{N} \to \mathbb{N} \to o$. The first argument to m represents a memory location and the second argument represents the value stored at that location.

$$t_m(M) = \begin{cases} \emptyset & \text{if } M = \emptyset \\ (m\ l\ v) \uplus t_m(M') & \text{if } M = (l, v) \cup M' \end{cases} \tag{5}$$

The ternary evaluation predicate e is defined in Figure 4 and has the type $\mathbb{R} \to \mathbb{N} \to o \to o$. This definition models the natural semantics rules given in Figure 1. Its first argument is the program expression to be evaluated, its second argument is an element from $\mathbb{N}$ representing the return value of the input program, and its third argument is a formula that must be proved in the memory left behind after the program expression has been evaluated. In this definition, explicit quantification has been removed for clarity. All capitalized terms occurring in the head of a program clause formula are universally quantified variables. All capitalized terms occurring exclusively in the body of a program clause formula are existentially quantified variables.

The e predicate relies heavily on a continuation-passing style[19] where the universally quantified variable C with type o is a continuation. The use of continuations allows a natural way to express the subsequent evaluation of program expression in potentially modified memory. For example, consider the program clause formula in Figure 4 modeling sequencing in the object system. As noted at the end of Subsection 2.1, one method for building a derivation would be by building derivations for the premises in a left-to-right order. We capture this method in this formula: the first program expression should be evaluated and this may result in modified memory, the second program expression should be modeled for evaluation in this modified memory. Therefore, we extend the continuation with an evaluation predicate for the second program expression.

For each natural semantics style rule given in section 2.1 there is a corresponding formula in Figure 4. A simple heuristic was followed to model each rule: modeled premises of a rule "extend" the continuation, becoming the body of a

$$\dfrac{\Gamma;\emptyset\vdash B_1 \quad \ldots \quad \Gamma;\emptyset\vdash B_n \quad \Gamma;\Delta_1\vdash C_1 \quad \ldots \quad \Gamma;\Delta_m\vdash C_m}{\Gamma, B;\Delta_1,\ldots,\Delta_m\vdash A}\,BC_u \qquad \dfrac{\Gamma;\emptyset\vdash B_1 \quad \ldots \quad \Gamma;\emptyset\vdash B_n \quad \Gamma;\Delta_1\vdash C_1 \quad \ldots \quad \Gamma;\Delta_m\vdash C_m}{\Gamma;\Delta_1,\ldots,\Delta_m, B\vdash A}\,BC_b$$

Figure 3. In the specification logic, these back chaining rules will replaces all left-introduction rules from Figure 2. Both have the proviso that $n, m \geq 0$ and $\langle\{B_1,\ldots,B_n\},\{C_1,\ldots,C_m\},A\rangle \in ||B||$

program clause formula. Continuation extension is done in left-to-right premise order. Finally, the conclusion of the rule will become the head of a program clause formula.

As we did in Section 2, we will overloaded the operators $+$, $-$, $>$, and $\leq$. Again, we associate the usual subtraction operation on natural numbers: $N_1 - N_2$ is 0 if N_1 is less than N_2.

Let Γ be a set exclusively containing the formulas from Figure 4. The evaluation relation $\langle E, M\rangle \rightsquigarrow (N, M')$ defined in Section 2.1 is translated to the sequent:

$$\Gamma; t_m(M) \vdash (e\ t_{\mathbb{R}}(E)\ N\ \top) \tag{6}$$

The use of $\top$ here "throws away" the memory resulting from this evaluation, i.e what is M' in the evaluation relation. If inspection of this memory is necessary, we may replace $\top$ with a goal formula. For example, if we wanted to inspect the value in memory stored at location 1 we could use the following sequent:

$$\Gamma; t_m(M) \vdash (e\ t_{\mathbb{R}}(E)\ N_1\ ((m\ 1\ N_2) \otimes \top)).$$

In our model of the object system, memory is accessed and modified using the subformula

$$(m\ N_1\ N_2) \otimes ((m\ N_1\ N_3) \multimap C)$$

where N_1, N_2, and N_3 have the type $\mathbb{N}$. When $N_2 = N_3$ the operation is a lookup, otherwise it is an update. If M is memory undefined at N_1 then a proof of this subformula must have the following structure due to the meta-theoretical results from Subsection 3.2 and our model.

$$\dfrac{\dfrac{}{\Gamma; (m\ N_1\ N_2) \vdash (m\ N_1\ N_2)}\,id \qquad \dfrac{\dfrac{\vdots}{\Gamma; t_m(M), (m\ N_1\ N_3) \vdash C}}{\Gamma; t_m(M) \vdash (m\ N_1\ N_3) \multimap C}\,\multimap R}{\Gamma; t_m(M), (m\ N_1\ N_2) \vdash (m\ N_1\ N_2) \otimes ((m\ N_1\ N_3) \multimap C)}\,\otimes R$$

Therefore, our treatment of state in our specification logic has an intuitive and logical reading of "remove the value N_2 at location N_1 and replace it with the value N_3".

4.2 Proofs as Computations

Consider the Ω derivation given in Subsection 2.2 for the relation

$$\langle 1 \leftarrow *2, M''\rangle \rightsquigarrow (M''(2), M''')$$

where

$$M'' = M[2 \mapsto M(0)][0 \mapsto M(1)]$$

and M is memory defined at $0, 1$ and 2. The sequent corresponding to this evaluation relation is

$$\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash \\ (e\ (1 \leftarrow *2)\ M''(0)\ \top)$$

where O is memory and for all $n \in \mathbb{N}$ if $n > 3$ then $O(n) = M(n)$, otherwise $O(n)$ is undefined. A proof of this sequent can be found in Figure 5. In this proof right introduction rules are omitted.

A mimicry of the derivation can be seen in this proof: for every rule instance that occurs in the derivation there is a corresponding BC_u instance with a principal program clause formula1 that models that derivation rule instance.

5. Reasoning about Properties of Imperative Programs Using the Model

In this section, we show that our model of the object system can be used to prove a similar property to what was shown in Subsection 2.3. As it was in Subsection 2.3, the property proven is trivial. However, the goal of this exercise is to demonstrate that the structure of the argument on the model follows very closely the structure of the argument from Subsection 2.3. In this sense, reasoning about properties of our model can be intuitive. This advantage when reasoning is a result of the mimicry exposed in Subsection 4.2.

5.1 Correctness as a Property of Proofs in the Model

We must model the Lemma 1 and Theorem 2 in the specification logic. The modeled lemma and theorem rely on the sum program from Equation 2, the term translation function from Equation 4, the memory translation function from Equation 5, the evaluation predicate e from Figure 4, and , tacitly, the relationship translation function from Equation 6. As defined in Subsection 4.1, the set Γ is exclusively inhabited by formulas from Figure 4.

Lemma 10 encodes Lemma 1 from Subsection 2.3 in the specification logic.

Lemma 10 (Total Correctness of $t_{\mathbb{R}}(Q)$)**.** $\forall N_1, N_2, M, M_0$ *if* $N_1, N_2 \in \mathbb{N}$ *and* $M = M_0[0 \mapsto N_2][1 \mapsto N_1]$ *then* $\exists N_3$, $N_3 \in \mathbb{N}$ *and* $\Gamma; t_m(M) \vdash (e\ t_{\mathbb{R}}(Q)\ 0\ (m\ 0\ N_3) \otimes \top)$ *and* $N_3 = N_2 + \sum_{i=0}^{N_1} i$

The value of a memory location was extracted by function application in the object system. In the encoding we retrieve the value from memory after program evaluation via a

$$\begin{array}{ll}
C & \multimap\ (e\ N\ N\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ (N_3 = N_1 + N_2 \otimes C))) & \multimap\ (e\ (\mathit{add}\ E_1\ E_2)\ N_3\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ (N_3 = N_1 - N_2 \otimes C))) & \multimap\ (e\ (\mathit{sub}\ E_1\ E_2)\ N_3\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ (N_1 > N_2 \otimes C))) & \multimap\ (e\ (\mathit{gt}\ E_1\ E_2)\ \mathit{sz}\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ (N_1 \leq N_2 \otimes C))) & \multimap\ (e\ (\mathit{gt}\ E_1\ E_2)\ z\ C) \\
(e\ E\ N_1\ ((m\ N_1\ N_2) \otimes ((m\ N_1\ N_2) \multimap C))) & \multimap\ (e\ (\mathit{get}\ E)\ N_2\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ ((m\ N_1\ N_3) \otimes ((m\ N_1\ N_2) \multimap C)))) & \multimap\ (e\ (\mathit{set}\ E_1\ E_2)\ N_2\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ C)) & \multimap\ (e\ (\mathit{seq}\ E_1\ E_2)\ N_2\ C) \\
(e\ E_1\ N_1\ (e\ E_2\ N_2\ (e\ (\mathit{wh}\ E_1\ E_2)\ C\ ))) \otimes N_1 > z & \multimap\ (e\ (\mathit{wh}\ E_1\ E_2)\ z\ C) \\
(e\ E_1\ N_1\ C) \otimes N_1 = z & \multimap\ (e\ (\mathit{wh}\ E_1\ E_2)\ z\ C)
\end{array}$$

Figure 4. The program clause formulas modeling the evaluation semantics given in section 2.1 in the specification logic.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{}\ \top R \atop \vdots}{\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash ((m\ 2\ M(0)) \otimes (m\ 2\ M(0))) \multimap ((m\ 1\ M(1)) \otimes ((m\ 1\ M(0)) \multimap \top))}}{\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash (e\ 2\ 2\ ((m\ 2\ M(0)) \otimes ((m\ 2\ M(0)) \multimap ((m\ 1\ M(1)) \otimes ((m\ 1\ M(0)) \multimap \top)))))}\ BC_U}{\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash (e\ (\mathit{get}\ 2)\ M(0)\ ((m\ 1\ M(1)) \otimes ((m\ 1\ M(0)) \multimap \top)))}\ BC_U}{\dfrac{\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash (e\ 1\ 1\ (e\ (\mathit{get}\ 2)\ M(0)\ ((m\ 1\ M(1)) \otimes ((m\ 1\ M(0)) \multimap \top))))}{\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash (e\ (\mathit{set}\ 1\ (\mathit{get}\ 2))\ M(0)\ \top)}\ BC_U}\ BC_U$$

Figure 5. A proof of the sequent $\Gamma; (m\ 0\ M(1)), (m\ 1\ M(1)), (m\ 2\ M(0)), t_m(O) \vdash (e\ (1 \leftarrow *2)\ M''(0)\ \top)$.

continuation formula. Specifically, in Lemma 10 that continuation formula is $(m\ 0\ N_3) \otimes \top$. This formula extracts only the value in memory at location 0. This is where we expect the result of program Q to be stored.

The encoding of Theorem 2 is similar. Observe that N_1 and N_2 are immediately initialized upon evaluation of P; this use of N_1 and N_2 is only meant to ensure that memory M is defined at locations 0 and 1.

Theorem 11 (Total Correctness of $t_{\mathbb{R}}(P)$)**.** *$\forall N_1, N_2, M, M_0$ if $N_1, N_2 \in \mathbb{N}$ and $M = M_0[0 \mapsto N_2][1 \mapsto N_1]$ then $\exists N_3$, $N_3 \in \mathbb{N}$ and $\Gamma; t_m(M) \vdash (e\ t_{\mathbb{R}}(P)\ 0\ (m\ 0\ N_3) \otimes \top)$ and $N_3 = \sum_{i=0}^{N} i$*

5.2 Reasoning about Proofs

Reasoning about the model is structured according to the reasoning structure in Subsection 2.3.

Proof of Lemma 10. This will be shown by induction on N_1. In the first case where $N_1 = 0$, the proof in Figure 6 can be constructed and we can conclude that $N_3 = N_2$. Therefore, we have that $N_3 = N_2 + \sum_{i=0}^{N_1} i$.

In the second case we assume that Lemma 10 holds if $M(1) < N_1$ and must show this lemma holds when $M(1) = N_1$; this is our inductive hypothesis. Proof analysis of the sequent

$$\Gamma; t_m(M'), (m\ 1\ M(1)), (m\ 0\ M(0)) \vdash \\ (e\ t_{\mathbb{R}}(Q)\ 0\ ((m\ 0\ N_3))$$

reveals that it suffices to build a proof for the sequent

$$\Gamma; (m\ 1\ (M(1) - 1)), (m\ 0\ (M(0) + M(1))), t_m(M') \\ \vdash (e\ t_{\mathbb{R}}(Q)\ 0\ ((m\ 0\ N_3) \otimes \top)).$$

We omit such a proof in our discussion here; it mimics the derivation for the second case given in Subsection 2.3, it is tedious, and, its construction is completely mechanizable in our specification logic. The inductive hypothesis yields this sequent. Additionally, by the inductive hypothesis, we have $N_4 = (M(0) + M(1)) + \sum_{i=0}^{M(1)-1} i$ for some $N_4 \in \mathbb{N}$. This is equivalent to $N_4 = M(0) + \sum_{i=0}^{M(1)} i$ and thus, $N_3 = N_4$. □

Proof of Theorem 11. We must prove

$$\Gamma; t_m(M) \vdash (e\ t_{\mathbb{R}}(P)\ 0\ (m\ 0\ N_3) \otimes \top)$$

and $N_3 = \sum_{i=0}^{N} i$. Proof analysis of this sequent reveals it is sufficient to prove the sequent

$$\Gamma; t_m(M'), (m\ 1\ N), (m\ 0\ 0) \vdash (e\ t_{\mathbb{R}}(Q)\ 0\ ((m\ 0\ N_3) \otimes \top))$$

$$
\dfrac{
  \dfrac{}{\Gamma; (m\ 1\ 0) \vdash (m\ 1\ 0)}\,BC_b
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{\Gamma; \vdash 0 \leq 0}\,\leq
        \qquad
        \dfrac{
          \dfrac{}{\Gamma; \vdash (0 <> 1)}\,\neq
          \qquad
          \dfrac{
            \dfrac{}{\Gamma; (m\ 0\ N_2) \vdash (m\ 0\ N_2)}\,BC_b
            \qquad
            \dfrac{}{\Gamma; (m\ 1\ 0), t_m(M') \vdash \top}\,\top R
          }{\Gamma; (m\ 1\ 0), t_m(M'), (m\ 0\ N_2) \vdash ((m\ 0\ N_2) \otimes \top)}\,\oplus R
        }{\Gamma; (m\ 1\ 0), t_m(M'), (m\ 0\ N_2) \vdash ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top))}\,\oplus R
      }{\Gamma; (m\ 1\ 0), t_m(M'), (m\ 0\ N_2) \vdash (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top)))}\,\oplus R
    }{\Gamma; (m\ 1\ 0), t_m(M'), (m\ 0\ N_2) \vdash (e\ 0\ 0\ (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top))))}\,BC_u
  }{\Gamma; t_m(M'), (m\ 0\ N_2) \vdash ((m\ 1\ 0) \multimap (e\ 0\ 0\ (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top)))))}\,\multimap R
}{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Gamma; t_m(M'), (m\ 1\ 0), (m\ 0\ N_2) \vdash ((m\ 1\ 0) \otimes ((m\ 1\ 0) \multimap (e\ 0\ 0\ (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top))))))}{\Gamma; t_m(M'), (m\ 1\ 0), (m\ 0\ N_2) \vdash (e\ 1\ 1\ ((m\ 1\ 0) \otimes ((m\ 1\ 0) \multimap (e\ 0\ 0\ (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top)))))))}\,BC_u
}{\Gamma; t_m(M'), (m\ 1\ 0), (m\ 0\ N_2) \vdash (e\ (get\ 1)\ 0\ (e\ 0\ 0\ (0 \leq 0 \otimes ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top)))))}\,BC_u
}{\Gamma; t_m(M'), (m\ 1\ 0), (m\ 0\ N_2) \vdash (e\ (gt\ (get\ 1)\ 0)\ 0\ ((0 <> 1) \otimes ((m\ 0\ N_2) \otimes \top)))}\,BC_u
}{\Gamma; t_m(M'), (m\ 1\ 0), (m\ 0\ N_2) \vdash (e\ (wh\ (gt\ (get\ 1)\ 0)\ (seq\ (set\ 0\ (add\ (get\ 0)\ (get\ 1)))\ (set\ 1\ (sub\ (get\ 1)\ 1))))\ 0\ ((m\ 0\ N_2) \otimes \top))}\,BC_u
}\,\oplus R
$$

Figure 6. A proof of the judgment $\Gamma; t_m(M) \vdash (e\ \ t_{\mathbb{R}}(Q)\ \ 0\ \ (m\ \ 0\ \ N_3) \otimes \top)$ where $N_1, N_2 \in \mathbb{N}$, $N_1 = 0$, $M = M_0[0 \mapsto N_2][1 \mapsto N_1]$, and M' is a partial function undefined at 0,1 and equal to M_o otherwise.

. We have both by Lemma 10. □

5.3 Extracting Properties from the Model

We would like to extract Lemma 10 and Theorem 11 into the object system. In general, doing so requires some confidence that the extracted property is meaningful in the object system. Such confidence is typically acquired through an informal adequacy argument [8].

The adequacy of an encoding can be shown by giving a bijective translation function from the object system to the encoding. There are complexities in providing such a translation for our encoding; in the object system, memory is a term while in the encoding memory is formula. How such a translation can be given is left to future work.

6. Conclusion

We have considered in this paper the possibility of formalizing the process of reasoning about properties of imperative programs. Towards this end, we have described a specification logic that can transparently model imperative programming languages with semantics defined in an natural semantics style. An important aspect of this specification logic is that its proof relation can be restructured so as to yield derivations that closely resemble the ones that may be constructed in the original natural semantics style encodings of object systems. We have illustrated how this characteristic can be exploited in reasoning about the properties of the object systems. In our example, we have used an informal style of reasoning over specification logic derivations. However, we believe that this reasoning process can be formalizing and we are examining this aspect in ongoing work. In particular, we are exploring the idea of using a two-level logic approach [6, 14] that has been successfully exploited in conjunction with an intuitionistic specification logic in the Abella system [6]. In this approach, we encode a specification logic via its derivability relation within a rich "reasoning" logic: by using the capabilities of the reasoning logic, we then obtain the ability to prove properties about derivations in the specification logic. One of our immediate goals is to accommodate a linear specification logic within the same reasoning logic that underlies Abella, thereby producing a variant of Abella that supports the development of formal arguments related to systems oriented around resource usage. Once we have an implementation of such a system at hand, the next step would be to use it to formalize the kinds of arguments we have presented in this paper.

In addition to actually implementing the ideas we have discussed in this paper within a formal system, we must also extend them so that we can reason about a larger, more realistic collection of programs. The imperative programs that we have considered in this paper use programming language constructs permitting non-termination and memory manipulations, i.e. lookup and update. In essence, we have demonstrated that our approach can be effective when reasoning about properties of basic imperative programs lacking pointers (because memory values were never used in lookups) or dynamic allocation. Going forward, we would like to examine two particular kinds of extensions to this work.

The language chosen in this paper does not permit complex notions of data, dynamic memory allocation, or functional aspects. It does permit references but the imperative program analyzed does not use them. A more relevant language to model would be a subset of SML [17] excluding data-type definitions and the module subsystem. This subset would not make modeling evaluation semantics much more complex. For example, memory allocation can be treated naturally using universal quantifiers. We conjecture that such changes will not alter the intuitive nature of reasoning.

The program chosen and its correctness property is trivial. Programs in common use among other researchers concerned with reasoning about imperative programs are linked list (singly or doubly) manipulation programs and implementations of the Schorr-Waite algorithm[1]. Additionally, properties of programs using references can be particularly difficult to reason about due to aliasing. Aliasing occurs when a location can be accessed in two different ways. For example, the program

$$1 \leftarrow 0; 2 \leftarrow 1; 3 \leftarrow 1; *2 \leftarrow 4; **3$$

is one where aliasing occurs; the last two program expressions will update and lookup, respectively, location 1. Reasoning in basic Hoare logic is unsound when programs containing aliasing are considered. Hoare logic can be extended such that reasoning about pointers is technically feasible but complex[18]. We have an inchoate idea that a treatment of aliasing should not require major changes to our specification logic because locations are not named. Therefore, references to aliased data is explicit. How this treatment will affect reasoning intuitions remains unclear.

Finally, we must better understand the connections between our approach and others such as ones using Hoare and separation based logic[3, 11, 15, 18], pointer assertion logic[10], parametric shape analysis[13], and aliasing logic[2].

References

[1] R. Bornat. Proving pointer programs in hoare logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67727-7.

[2] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In R. Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 344–360. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22791-5.

[3] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. In S. Artemov and A. Nerode, editors, *Logical Foundations of Computer Science*, volume 4514 of *Lecture Notes in Computer Science*, pages 100–114. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-72732-3.

[4] A. Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.

[5] A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008. URL `http://arxiv.org/abs/0803.2305`.

[6] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012. . URL `http://arxiv.org/abs/0911.2993`.

[7] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50: 1–102, 1987.

[8] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17 (4–5):613–673, July 2007.

[9] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *6th Symp. on Logic in Computer Science*, pages 32–42, Amsterdam, July 1991.

[10] J. L. Jensen, M. E. Jørgensen, M. I. Schwartzbach, and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 226–234, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. . URL `http://doi.acm.org/10.1145/258915.258936`.

[11] L. Jia and D. Walker. Ilc: A foundation for automated reasoning about pointer programs. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 131–145. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33095-0.

[12] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, Mar. 1987.

[13] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In J. Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67668-3.

[14] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[15] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40559-7.

[16] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[18] P. OHearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42554-0.

[19] G. J. Sussman and G. L. S. Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.