

# 1 Sound Notional Machines

2 ANONYMOUS AUTHOR(S)

3 A notional machine is a pedagogical device that abstracts away details of the semantics of a programming  
4 language to focus on some aspects of interest. A notional machine should be *sound*: it should be consistent  
5 with the corresponding programming language, and it should be a proper abstraction. This reduces the risk  
6 of it introducing misconceptions in education. Despite being widely used in computer science education,  
7 notional machines are usually not evaluated with respect to their soundness. To address this problem, we first  
8 introduce a formal definition of soundness for notional machines. The definition is based on the construction  
9 of a commutative diagram that relates the notional machine with the aspect of the programming language  
10 under its focus. Derived from this formalism, we present a methodology for constructing sound notional  
11 machines, which we demonstrate by applying it to a series of small case studies. We also show how the  
12 same formalism can be used to analyze existing notional machines and find inconsistencies in them as well  
13 as propose solutions to these inconsistencies. The work establishes a firmer ground for research in notional  
14 machines by serving as a framework to reason about them.

15  
16 CCS Concepts: • Social and professional topics → Computing education; • Software and its engineering  
17 → Formal language definitions; Context specific languages.

18 Additional Key Words and Phrases: notional machines, programming education

## 19 1 INTRODUCTION

20 Learning to program involves learning how to express a program in a programming language, but  
21 also learning what the semantics of such a program is. For novices, the semantics of a program  
22 is often not obviously apparent from the program itself. Instructors then often use a *notional*  
23 *machine* [Fincher et al. 2020] to help teach some particular aspect of programs and programming  
24 languages, and also to assess students’ understanding of said aspect. This aspect is the notional  
25 machine’s focus. For example, showing expressions as trees (the EXPTREE notional machine), as  
26 depicted in Figure 1, brings out the internal structure of lambda-calculus expressions [Marceau et al.  
27 2011] and can help to explain the step-by-step evaluation of such expressions. Notional machines  
28 are used widely in computer science education; Fincher et al. [2020] interviewed computer science  
29 teachers to build up a collection of over 50 notional machines<sup>1</sup>.

### 31 32 1.1 Unsound Notional Machines

33 Given the extensive use of notional machines, and their intended use as devices to help students  
34 when learning, it is important to look at their quality. To begin with, one should make sure that the  
35 notional machine is *sound*: it is faithful to the aspect of programs it is meant to focus on. Anecdotal  
36 evidence of using unsound representations in education goes back a long way. In 1960, education  
37 pioneer Jerome Bruner wrote that “the task of teaching a subject to a child at any particular age is  
38 one of representing the structure of that subject in terms of the child’s way of viewing things”, that  
39 this “task can be thought of as one of translation”, but that ideas have to be “represented *honestly*  
40 [...] in the thought forms of children” [Bruner 1960]. Bruner’s use of the term “honestly” can be  
41 seen as a call for soundness of such representations. Decades later, Richard Feynman eloquently  
42 stated [Feynman 1985], after reviewing “seventeen feet” of new mathematics schoolbooks for the  
43 California State Curriculum Commission:

44 [The books] would try to be rigorous, but they would use examples (like automobiles  
45 in the street for “sets”) which were almost OK, but in which there were always some  
46 subtleties. The definitions weren’t accurate. Everything was a little bit ambiguous.

47  
48 <sup>1</sup><https://notionalmachines.github.io/notional-machines.html>

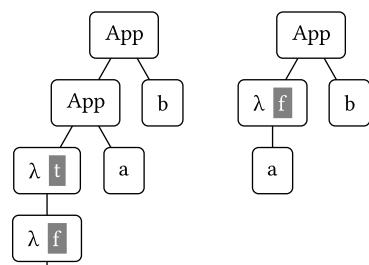
	Step 0	Step 1	Step 2
51	$(\lambda t.(\lambda f.t))\ a\ b$	$(\lambda f.a)\ b$	a
52			
53			
54			
55			
56			
57			
58			
59			
60			
61			
62			

Fig. 1. Evaluation of  $(\lambda t.(\lambda f.t))\ a\ b$  in programming language (top) and ExPTREE notational machine (bottom).

Ambiguously specified notional machines and notional machines with imperfect analogies to programming concepts are a problem. Educators may mischaracterize language features and students may end up with misconceptions [Chiodini et al. 2021] instead of profoundly understanding the language.

For example, Fincher et al. [2020] describe the “Array as Row of Spaces in Parking Lot” notional machine. Figure 2 shows part of their card summarizing it. Notice the parallels between the programming language (PL) and the notional machine (NM). Consider Java, a language commonly used in programming courses. In Java, when an array of objects is allocated, all its slots contain **null**, which means these slots don’t contain a reference to any object. This would be reasonably represented in the notional machine as an empty parking lot. But if instead of an array of objects, we have an array of **ints**, for example, then when we instantiate an array, all its slots contain  $\emptyset$ , which is not the absence of a number but a number like any other. A student could also reasonably question whether one can park a car in a slot that is already occupied by another car, or whether one has to remove a car from a spot to park another car in the same spot. In fact, the authors point on [...] how well that models the semantics of

## 1.2 Soundness of Notional Machines via Simulation

To avoid these issues, we need to make sure that a notional machine is indeed an accurate abstraction. Although the more general view of notional machines as “pedagogic devices to assist

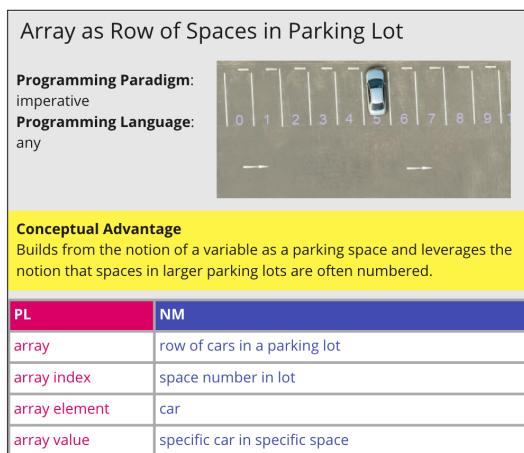


Fig. 2. The “Array as Row of Spaces in Parking Lot” notional machine captured by Fincher et al. [2020].

the understanding of some aspect of programs or programming” [Fincher et al. 2020] makes no direct reference to programming languages, programs are expressed in programming languages so we will look at these “aspects of programs” through the lens of how they are realized by some programming language. If a notional machine represents a part of the operational semantics of a programming language, for example, then this representation should be sound, in the sense that steps in the notional machine correspond to steps in the operational semantics of the programming language.

Showing the soundness of a notional machine amounts to demonstrating that the notional machine *simulates* (in the sense described by Milner [1971]) the aspect of programs under the focus of the notional machine. This property can be given in the form of a commutative diagram. Milner’s simulation was also used by Hoare [1972] to establish a definition of the correctness of an ‘abstract’ data type representation with respect to its corresponding ‘concrete’ data type representation. This interpretation of simulation also captures the relationship between a notional machine and the underlying programming language because a notional machine is indeed an *abstraction* of some aspect of interest.

### 1.3 Contributions

This paper makes the following contributions:

- A formal definition of sound notional machine based on simulation (Section 2.1.2).
- A methodology for designing notional machines that are sound by construction.

It consists of deriving part of the notional machine by leveraging the relationship between the abstract representation of the notional machine and the abstract syntax of the programming language under its focus.

We demonstrate the methodology by applying it to a combination of various small programming languages, notional machines, and aspects of programming language semantics (Section 2).

- A methodology for analyzing notional machines with respect to their soundness.

It consists of modelling the notional machine and the aspect of the programming language under its focus, relating them via simulation.

We demonstrate the methodology by analyzing existing notional machines, sometimes pointing out unsoundnesses, and suggesting directions for improvement (Section 3).

A brief discussion about the distinction between the abstract and concrete representations of a notional machine follows in Section 4. We then evaluate, in Section 5, the entire framework by comparing the notional machines that appeared in Section 2 and Section 3 to an existing dataset of 37 notional machines. We classify the notional machines in the dataset according to various dimensions and show that the notional machines we analyzed are representative of the design space of notional machines used in practice. Section 6 discusses related work and Section 7 concludes.

## 2 DESIGNING SOUND NOTIONAL MACHINES

We can only begin to talk about the soundness of a notional machine if we have a formal description of the programming language the notional machine is focused on. We use a set of small programming languages with well-known formalizations described in Pierce’s Types and Programming Languages (TAPL) book [Pierce 2002]. The languages are used to explore different aspects of programming language semantics. Table 1 shows the notional machines we use in this section as well as the corresponding programming language and aspect of the semantics of the programming language that the notional machine focuses on. We use the first example also to introduce the definition of soundness for notional machines.

Section	Notional Machine	Programming Language	Focus
2.1	EXPTREE	UNTYPEDLAMBDA	Evaluation
2.2	EXPTUTORDIAGRAM	UNTYPEDLAMBDA	Evaluation
2.3	TAPLMEMORYDIAGRAM	TYPEDLAMBDAREF	Evaluation (Refs)
2.4	TYPEDEXPTUTORDIAGRAM	TYPEDARITH	Types

Table 1. Notional machines, programming languages, and aspects of focus used in Section 2.

$$\begin{array}{ccc}
 A_{NM} & \xrightarrow{f_{NM}} & B_{NM} \\
 \alpha_A \uparrow & & \uparrow \alpha_B \\
 A_{PL} & \xrightarrow{f_{PL}} & B_{PL}
 \end{array}$$

$$\alpha_B \circ f_{PL} \equiv f_{NM} \circ \alpha_A \quad (1)$$

Fig. 3. Soundness condition for notional machines shown as a commutative diagram and in algebraic form.

We model each programming language and notional machine in Haskell. The models are executable, so they include implementations of the programming languages (including parsers, interpreters, and type-checkers), the notional machines, and the relationship between them<sup>2</sup>. The soundness proofs presented in this section are done using equational reasoning [Bird 1989; Gibbons 2002].

## 2.1 Isomorphic Notional Machines

As a first straightforward example, let's look at a notional machine for teaching how evaluation works in the untyped lambda-calculus (we will refer to this language as UNTYPEDLAMBDA<sup>3</sup>). While most research papers discuss the lambda-calculus using its textual representation, textbooks sometimes illustrate it using tree diagrams [Pierce 2002, p. 54]. We use this as an opportunity to define a simple notional machine which we call EXPTREE.

**2.1.1 Illustrative Example.** Figure 1 uses EXPTREE to demonstrate the evaluation of a specific lambda expression, which happens in two reduction steps. The top of the figure shows the terms in the traditional textual representation of the programming language, while the bottom shows the terms as a tree.

**2.1.2 Soundness via Commutative Diagrams.** In general, a notional machine is sound if the diagram in Figure 3 commutes. We call the commutativity of this diagram the *soundness condition* for a notional machine. In this diagram, the vertices are types and the edges are functions. We will explain the diagram while instantiating it for this illustrative example (the result of this instantiation is shown in Figure 4).

<sup>2</sup>The artifact containing a superset of the examples in this paper is at ANONYMOUS.

<sup>3</sup>The syntax and reduction rules for UNTYPEDLAMBDA are reproduced in Appendix A.1.

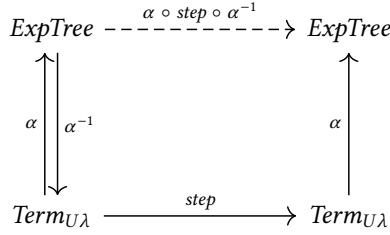


Fig. 4. Instantiation of the commutative diagram in Figure 3 for the notional machine  $\text{ExpTree}$  and the programming language  $\text{UNTYPEDLAMBDA}$ .

The bottom layer  $(A_{PL}, f_{PL}, B_{PL})$  represents the aspect of a programming language<sup>4</sup> we want to focus on.  $A_{PL}$  is an abstract representation of a program in that language. In our example, that is the abstract syntax of  $\text{UNTYPEDLAMBDA}$  (given by the type  $\text{Term}_{U\lambda}$ ). The function  $f_{PL}$  is an operation the notional machine is focusing on. In our example, that would be  $\text{step}$ , a function that performs a reduction step in the evaluation of a program according to the operational semantics of the language, which in this case also produces a value of type  $\text{Term}_{U\lambda}$ .

The top layer of the diagram  $(A_{NM}, f_{NM}, B_{NM})$  represents the notional machine.  $A_{NM}$  is an abstract representation of the notional machine (its abstract syntax). In our simple example, that is a type  $\text{ExpTree}$  trivially isomorphic to  $\text{Term}_{U\lambda}$  via a simple renaming of constructors. The function  $f_{NM}$  is an operation on the notional machine which should correspond to  $f_{PL}$ . Connecting the bottom layer to the top layer, there are the functions  $\alpha_A$  and  $\alpha_B$  from the abstract representation of a program in the programming language to the abstract representation of the notional machine.  $\alpha$  is also called an abstraction function.

*Definition 2.1.* Given the notional machine  $(A_{NM}, B_{NM}, f_{NM} : A_{NM} \rightarrow B_{NM})$ , focused on the aspect of a programming language given by  $(A_{PL}, B_{PL}, f_{PL} : A_{PL} \rightarrow B_{PL})$ , the notional machine is *sound* iff there exist two functions  $\alpha_A : A_{PL} \rightarrow A_{NM}$  and  $\alpha_B : B_{PL} \rightarrow B_{NM}$  such that  $\alpha_B \circ f_{PL} \equiv f_{NM} \circ \alpha_A$ .

If the abstract representation of the programming language  $(A_{PL})$  is isomorphic to the abstract representation of the notional machine  $(A_{NM})$ , we can construct an inverse mapping  $\alpha_A^{-1}$  such that  $\alpha_A^{-1} \circ \alpha_A \equiv id \equiv \alpha_A \circ \alpha_A^{-1}$ . In that case, we can always define a correct-by-construction operation  $f_{NM}$  on  $A_{NM}$  in terms of an operation  $f_{PL}$  on  $A_{PL}$ :

$$\begin{aligned} f_{NM} &:: A_{NM} \rightarrow B_{NM} \\ f_{NM} &= \alpha_B \circ f_{PL} \circ \alpha_A^{-1} \end{aligned}$$

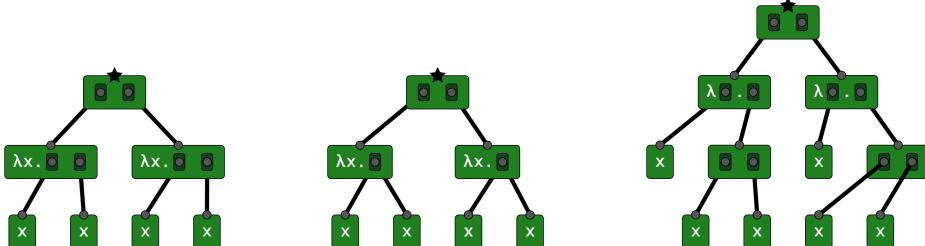
In such cases, the diagram always commutes and therefore the notional machine is sound:

$$f_{NM} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \circ \alpha_A^{-1} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \quad (2)$$

Instantiating the commutative diagram for  $\text{ExpTree}$  and  $\text{UNTYPEDLAMBDA}$  yields the diagram in Figure 4. A dashed line indicates a function that is implemented in terms of the other functions in the diagram and/or standard primitives.

<sup>4</sup>Although we refer to the bottom layer of the diagram as the programming language layer and we restrict ourselves to analyzing aspects of the syntax and semantics of programming languages, for which we have well-established formalizations, that is not an intrinsic restriction of the approach. In principle, the bottom level of the diagram can be whatever aspects of programs or programming the notional machine is focused on.

246  
247  
248  
249  
250  
251  
252  
253  
254  
255

 $(\lambda x. x \ x) \ (\lambda x. x \ x)$ 


256 Fig. 5. The omega combinator in UNTYPEDLAMBDA (top) and (incorrect) representations in ExprTUTORDIAGRAM  
257 notional machine (bottom).

258

259

260 We call these isomorphic notional machines because they are isomorphic to the aspect of the  
261 programming language they focus on (a condition sometimes called *strong simulation* [Milner  
262 1971]). Of course that's a rather strong condition and not every notional machine is isomorphic  
263 so throughout the next sections we will move further away from this simple example, arriving at  
264 other instantiations of this commutative diagram.

265

266

## 2.2 Monomorphic Notional Machines

267

Notional machines can also serve as the basis for so-called “visual program simulation” [Sorva et al.  
268 2013] activities, where students manually construct representations of the program execution. This  
269 effort often is supported by tools, such as interactive diagram editors, that scaffold the student’s  
270 activity. Obviously, instructors will want to see their students creating correct representations.  
271 However, to prevent students from blindly following a path to a solution prescribed by the tool, the  
272 visual program simulation environment should also allow *incorrect* representations.

273

274

ExpressionTutor<sup>5</sup> is such an educational tool to teach the structure, typing, and evaluation of  
expressions in programming courses. In ExpressionTutor, students can interactively construct  
expression tree diagrams given a source code expression. The tool is language agnostic so each  
node can be freely constructed (by the instructor or the student) to represent nodes of the abstract  
syntax tree of any language. Nodes can contain any number of holes that can be used to connect  
nodes to each other. Each hole corresponds to a place in an abstract syntax tree node where an  
expression would go. Nodes can be connected in a variety of ways, deliberately admitting incorrect  
structures that not only may not be valid abstract syntax trees of a given programming language  
but may not even be trees. Even the root node (labeled with a star) has to be explicitly labeled by  
the student, so it is not guaranteed to exist in every diagram.

283

284

We define the notional machine ExprTUTORDIAGRAM, which models the behavior of ExpressionTutor. The fact that ExpressionTutor allows students to construct incorrect expression tree diagrams means that the abstraction function  $\alpha$  is not bijective, as was the case of EXP TREE’s  $\alpha$ . Such incorrect diagrams do not correspond to programs, thus  $\alpha$  is deliberately not surjective.

287

288

289

290

291

292

293

294

**2.2.1 Illustrative Example.** Figure 5 uses ExprTUTORDIAGRAM to represent the omega combinator. The top shows the textual form on the level of the programming language. Below that are three different incorrect representations students could produce. The left tree collapses the applications (the terms  $x \ x$ ) into the enclosing lambda abstraction. The middle tree similarly does this, but it preserves the structure of the lambda abstraction node, while violating the well-formedness of the

<sup>5</sup>[expressiontutor.org](http://expressiontutor.org)

tree by plugging two children into the same hole. The right tree shows a different problem, where the definition of the name is pulled out of the lambda abstraction and shown as a name use.

**2.2.2 Commutative Diagram.** In general, if the mapping  $\alpha_A$ , from the abstract representation of the programming language ( $A_{PL}$ ) to the abstract representation of the notional machine ( $A_{NM}$ ), is an injective but non-surjective function, we can still define the operations on  $A_{NM}$  in terms of the operations on  $A_{PL}$ . For this we define a function  $\alpha_A^\circ : A_{NM} \rightarrow \text{Maybe } A_{PL}$  to be a left inverse of  $\alpha_A$  such that  $\alpha_A^\circ \circ \alpha_A \equiv \text{return}$  (we use *return* and *fmap* to refer to the *unit* and *map* operations on monads). Here we modeled the left inverse using a *Maybe* but another monad could be used, for example, to capture information about the values of type  $A_{NM}$  that do not have a corresponding value in  $A_{PL}$ . The top-right vertex of the square ( $B_{NM}$ ) in this case is the type  $\text{Maybe } B'_{NM}$  and the mapping  $\alpha_B$  can be implemented in terms of a mapping  $\alpha'_B : B_{PL} \rightarrow B'_{NM}$  like so:

$$\begin{aligned}\alpha_B &:: B_{PL} \rightarrow B_{NM} \\ \alpha_B &= \text{return} \circ \alpha'_B\end{aligned}$$

Using the left inverse  $\alpha_A^\circ$  and  $\alpha'_B$ , we define the operation on  $A_{NM}$  as follows:

$$\begin{aligned}f_{NM} &:: A_{NM} \rightarrow B_{NM} \\ f_{NM} &= \text{fmap } \alpha'_B \circ \text{fmap } f_{PL} \circ \alpha_A^\circ\end{aligned}$$

This square commutes like so:

$$\begin{array}{lll} f_{NM} \circ \alpha_A & & \alpha_B \circ f_{PL} \\ \equiv & \{ \text{definition of } f_{NM} \} & \equiv & \{ \text{definition of } \alpha_B \} \\ \text{fmap } \alpha'_B \circ \text{fmap } f_{PL} \circ \alpha_A^\circ \circ \alpha_A & & \text{return} \circ \alpha'_B \circ f_{PL} \\ \equiv & \{ \alpha_A^\circ \text{ is left inverse of } \alpha_A \} & \equiv & \{ \text{third monad law} \} \\ \text{fmap } \alpha'_B \circ \text{fmap } f_{PL} \circ \text{return} & & \text{fmap } \alpha'_B \circ \text{return} \circ f_{PL} \\ \equiv & \{ \text{third monad law} \} & & \end{array}$$

We can use this result to instantiate the commutative diagram of Figure 3 for *ExpTUTORDIAGRAM* and *UNTYPEDLAMBDA*, shown in Figure 6.  $A_{PL}$  is defined to be the type *ExpTutorDiagram*, which essentially implements a graph. Each node has a top plug and any number of holes, which contain plugs. Edges connect plugs. That allows for a lot of flexibility in the way nodes can be connected.

The ExpressionTutor tool is language agnostic but we can only talk about soundness of a notional machine with respect to some language and some aspect of that language. In this case, we say the tool implements a family of notional machines, each one for a given aspect of focus and a given programming language. We consider here a notional machine focused on evaluation and the programming language again *UNTYPEDLAMBDA* (denoted again by the type  $\text{Term}_{U\lambda}$ ), with  $f_{PL}$  equal to *step*.

The construction of a mapping  $\alpha :: \text{Term}_{U\lambda} \rightarrow \text{ExpTutorDiagram}$  is straightforward because a term  $t :: \text{Term}_{U\lambda}$  forms a tree and from it we can always construct a corresponding ExpressionTutor diagram  $d :: \text{ExpTutorDiagram}$  (a graph). For each possible term in  $\text{Term}_{U\lambda}$ , we need to define a pattern for the content of the corresponding *ExpTutorDiagram* node which will help the student identify the kind of node. The construction of the left inverse mapping  $\alpha^\circ :: \text{ExpTutorDiagram} \rightarrow \text{Maybe } \text{Term}_{U\lambda}$  requires more care. We need to make sure that the diagram forms a proper tree and that the pattern formed by the contents of each *ExpTutorDiagram* node corresponds to a possible  $\text{Term}_{U\lambda}$  node, besides making sure that they are connected in a way that indeed corresponds to a valid  $\text{Term}_{U\lambda}$  tree.

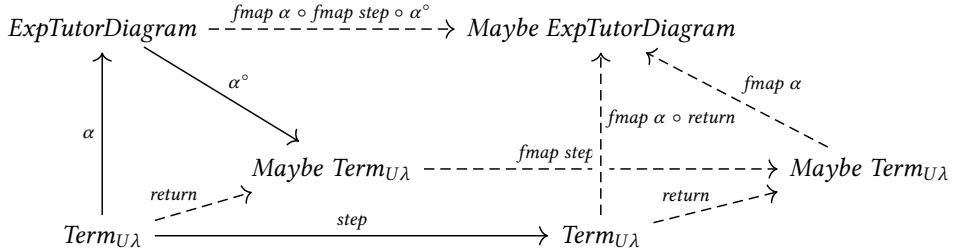


Fig. 6. Instantiation of the commutative diagram in Figure 3 for the notional machine `ExpTutorDiagram`

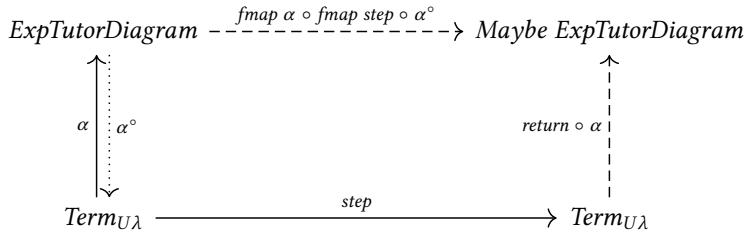


Fig. 7. Simplified version of the commutative diagram for `ExpTutorDiagram` shown in Figure 6.

In the next section, we construct another commutative diagram where  $f_{NM}$  is defined using  $f_{PL}$  and a left inverse mapping  $\alpha_A^\circ$ . To emphasize that point and simplify the diagrams, we will depict the left inverse in the diagram as a dotted line pointing from  $A_{NM}$  to  $A_{PL}$  (even though  $\alpha_A^\circ$  is of type  $A_{NM} \rightarrow \text{Maybe } A_{PL}$  and not  $A_{NM} \rightarrow A_{PL}$ ) and omit the path via  $\text{Maybe } A_{PL}$  as shown in Figure 7.

We call these monomorphic notional machines because there is a monomorphism (injective homomorphism) between the notional machine and the aspect of the programming language it focuses on. This is the case here by design, to allow students to make mistakes by constructing wrong diagrams that don't correspond to programs. In general, this will be the case whenever there are values of  $A_{NM}$  (the abstract syntax of the notional machine) that have no correspondence in the abstract representation of the language ( $A_{PL}$ ). That's often the case in memory diagrams [Dalton and Krehling 2010; Dragon and Dickson 2016; Holliday and Luginbuhl 2004] (notional machines used to show the relationship between programs and memory) because they typically allow for the representation of memory states that cannot be produced by legal programs. We show an example of such a notional machine in the next section.

### 2.3 A Monomorphic Notional Machine to Reason About State

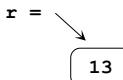
A common use of notional machines is in the context of reasoning about state<sup>6</sup>. An example of the use of a visual notation to represent state can also be found in TAPL [Pierce 2002, p. 155]. In Chapter 13 (“References”), the book extends the simply typed lambda-calculus with references (a language we will refer to as `TYPEDLAMBDAREF`<sup>7</sup>). It explains references and aliasing by introducing a visual notation to highlight the difference between a *reference* and the *cell* in the store that is

<sup>6</sup>29 of the 57 notional machines presented on <https://notionalmachines.github.io/notional-machines.html> at the time of this writing refer to the concept of *variable* or *array element*.

<sup>7</sup>The syntax and reduction rules for `TYPEDLAMBDAREF` are reproduced in Appendix A.3.

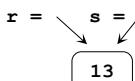
393 pointed to by that reference. We will refer to this notation, which we will develop into a notional  
 394 machine, as TAPLMEMORYDIAGRAM. In this notation, references are represented as arrows and  
 395 cells are represented as rounded rectangles containing the representation of the value contained in  
 396 the cell. Before designing the notional machine, we need to see the context in which this notation  
 397 is used in the book.

398 The book first uses this notation to explain the effect of creating a reference. It shows that when  
 399 we reduce the term `ref 13` we obtain a reference (a store location) to a store cell containing 13.  
 400 The book then represents the result of binding the name `r` to such a reference with the following  
 401 diagram:



402 In the book, this operation is written as `r = ref 13`, but as we will see in the next Section, this  
 403 form of name binding (name = term) exists only in a REPL-like context which is not part of the  
 404 language.

405 The book continues explaining that we can “make a copy of `r`” by binding its value to another  
 406 variable `s` (with `s = r`) and shows the resulting diagram:  
 407



408 The book then explains that one can verify that both names refer to the same cell by *assigning* a  
 409 new value to `s` and reading this value using `r` (for example, the term `s := 82; !r` would evaluate to  
 410 82). Right after, the book suggests to the reader an exercise to “draw a similar diagram showing the  
 411 effects of evaluating the expressions `a = {ref 0, ref 0}` and `b = (λx:Ref Nat.{x, x}) (ref 0)`.  
 412 Although we understand informally the use of this diagram in this context, how can we know what  
 413 a correct diagram would be in general for any given program? This is what we aim to achieve by  
 414 designing a notional machine based on this notation.

415 Let’s see how we would turn that kind of diagram into a sound notional machine. We want  
 416 to construct a commutative diagram where  $A_{PL}$  is an abstract representation of the state of a  
 417 TYPEDLAMBDAREF program execution,  $A_{NM}$  is an abstract representation of the diagram presented  
 418 in the book, and  $f_{PL}$  is an operation that affects the state of the store during program execution.

419 In a first attempt, let’s choose  $f_{PL}$  to be an evaluation step and  $A_{PL}$  to be modeled as close as  
 420 possible to the presentation of a TYPEDLAMBDAREF program as described in the book. In that  
 421 case,  $A_{PL}$  is the program’s abstract syntax tree together with a *store*, a mapping from a location (a  
 422 reference) to a value.

423 **2.3.1 Problem: Beyond the Language.** The first challenge is that the name-binding mechanism  
 424 used in the examples above (written as name = term) exists only in a REPL-like context in the  
 425 book used for the convenience of referring to terms by name. It is actually not part of the language  
 426 (TYPEDLAMBDAREF) so it is not present in this representation of  $A_{PL}$  and as a result it cannot be  
 427 mapped to  $A_{NM}$  (the notional machine). We will avoid this problem by avoiding this name-binding  
 428 notation entirely and writing corresponding examples fully in the language. The only mechanism  
 429 actually in the language to bind names is by applying a lambda to a term. Let’s see how we can write  
 430 a term to express the behavior described in the example the book uses to introduce the diagram  
 431 (shown earlier), where we:

- 432 (1) Bind `r` to the result of evaluating `ref 13`
- 433 (2) Bind `s` to the result of evaluating `r`

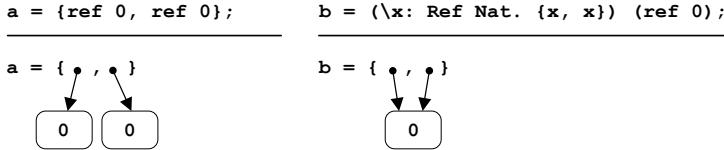


Fig. 8. TAPL MEMORY DIAGRAM for TYPED LAMBDA REF for TAPL Exercise 13.1.1

- (3) Assign the new value 82 to s  
 (4) Read this new value using r

Using only the constructs in the language, we express this with the following term:

$$(\lambda r: \text{Ref Nat}. (\lambda s: \text{Ref Nat}. s := 82; !r) r) \text{ (ref 13)}$$

**2.3.2 Problem: Direct Substitution.** The problem now is that if we model  $A_{PL}$  and evaluation as described in the book, the result of reducing a term  $(\lambda x. t_1) t_2$  is the term obtained by replacing all free occurrences of  $x$  in  $t_1$  by  $t_2$  (modulo alpha-conversion), so we don't actually keep track of name binding information. What we have in  $A_{PL}$  at each step is an abstract syntax tree and a store, but we have no information about which names are bound to which values because the names were already substituted in the abstract syntax tree. That would be enough to do Exercise 13.1.1, for example, whose solution is shown in Figure 8. But the absence of explicit information mapping names to values makes it less suitable to talk about aliasing, because even though a term may contain, at any given point during evaluation, multiple occurrences of the same location, it is not possible to know if these locations correspond to different names, and one may need to trace several reduction steps back to find out when a name was substituted by a location.

We need to change  $A_{PL}$  and step to capture this information, keeping not only a store but an explicit name environment that maps names to values, and only substituting the corresponding value when we evaluate a variable. Like the definition of application in terms of substitution, we have to be careful to avoid variable capture by generating fresh names when needed.

**2.3.3 Illustrative Example.** Figure 9 shows two variations of the notional machine being used to explain the evaluation of the term we had described before. It shows the state after each reduction step, on the left without an explicit name environment and on the right with an explicit name environment. Between each step, a line with the name of the applied reduction rule is shown. Notice that the representation with a name environment requires extra name lookup steps.

In both variations, the representation of the program (the term) being evaluated appears first (with gray background). Each term is actually an abstract syntax tree, which we represent here, like in the book, with a linearized textual form. Location terms are represented as arrows starting from where they appear in the abstract syntax tree and ending in the store cell they refer to.

The naming environment is shown as a table from variable names to terms. Store cells also contain terms. This means the textual representation of terms that appear both inside name environments and inside cells may also contain arrows to other cells.

**2.3.4 Commutative Diagram.** Similar to the abstract representation of the program execution, the abstract representation of the notional machine contains three parts: one for the program being evaluated, one for the name environment, and one for the store.

## Sound Notional Machines

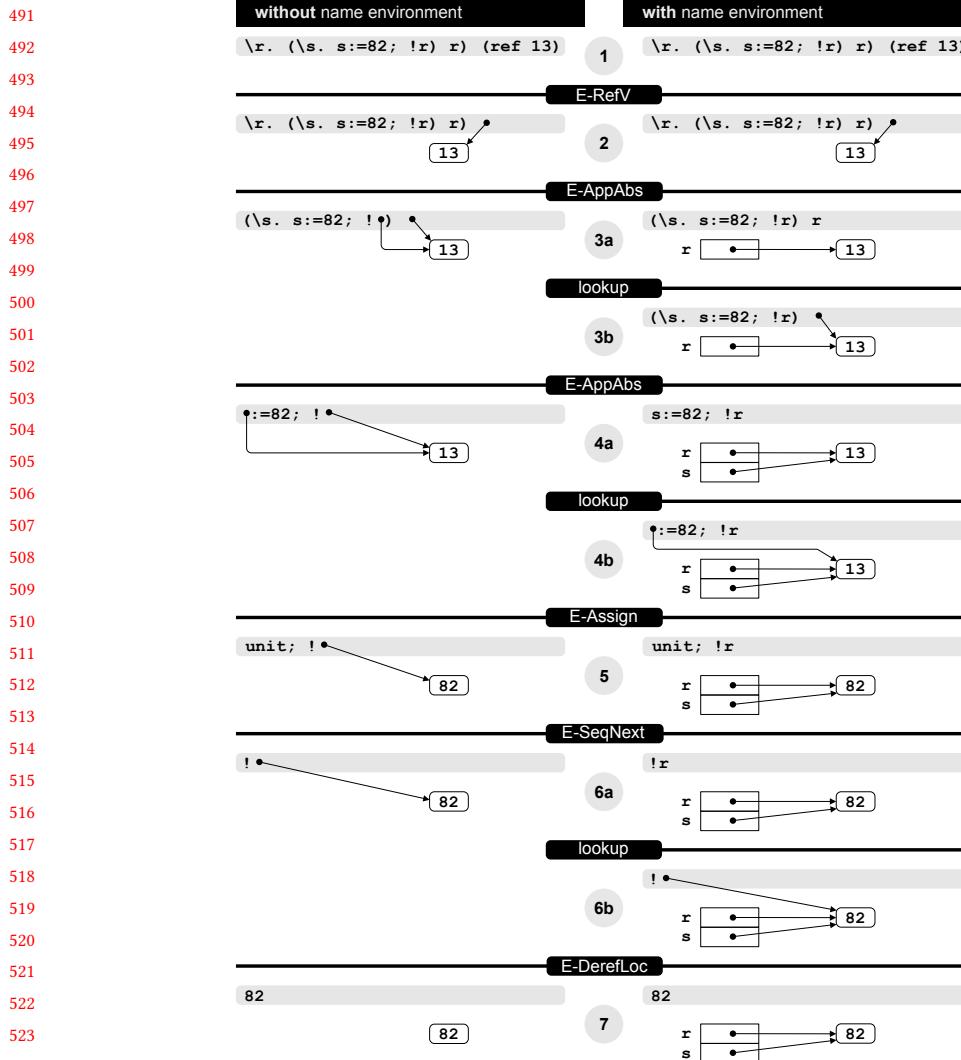
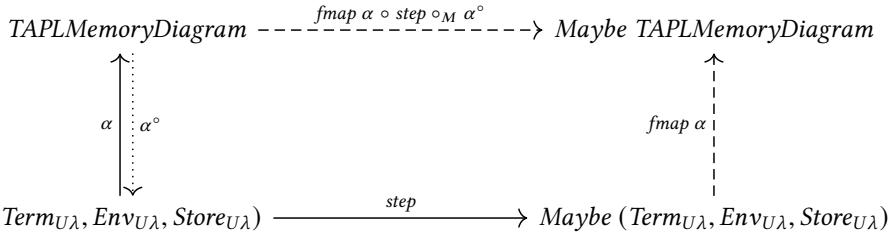


Fig. 9. Trace of  $(\lambda r: \text{Ref Nat}. (\lambda s: \text{Ref Nat}. s := 82; !r) r) \text{ (ref 13)}$  in TAPLMEMORYDIAGRAM for TYPEDLAMBDAREF

```

530  data TAPLMemoryDiagram l = TAPLMemoryDiagram {
531    memDiaTerm :: DTerm l,
532    memDiaNameEnv :: Map Name (DTerm l),
533    memDiaStore :: Map (DLocation l) (DTerm l)
534  }
535  data DTerm l = Leaf String
536    | Branch [DTerm l]
537    | TLoc (DLocation l)
538
539

```



549 Fig. 10. Instantiation of the commutative diagram in Figure 3 for the notional machine `TAPLMEMORYDIAGRAM`  
550 and the programming language `TYPEDLAMBDAREF`.

551  
552     The type *DLocation* corresponds to arrow destinations (arrow endpoints). A term is represented  
553     as a rose tree of *Strings* augmented with a case for location.  
554

555     The concrete representation of a *DTerm* can be in linearized text form or as a tree akin to that  
556     shown in Section 2.1. The representation of the nodes in a *DTerm* tree that are *TLoc* are shown as  
557     arrow starting points. These arrows end in the cell corresponding to the *DLocation* in each *TLoc*.  
558     The concrete representation of the store relates the visual position of each cell with the *DLocation*  
559     of each cell. That leads to the commutative diagram in Figure 10, where we use the symbol  $\circ_M$  to  
560     denote monadic function composition (the fish operator `<=<` in Haskell).

#### 561 2.4 A Monomorphic Notional Machine to Reason About Types

562 So far we have seen examples of commutative diagrams where  $f_{PL}$  is *step* (a function that performs a  
563 reduction step) but in principle,  $f_{PL}$  could be any operation on  $A_{PL}$  that is the focus of a given notional  
564 machine. Let's look at an example of notional machine where we do not focus on evaluating but on  
565 typing an expression. The language this notional machine focuses on is `TYPEDARITH`<sup>8</sup>, a language  
566 of typed arithmetic expression, which is the simplest typed language introduced in TAPL [Pierce  
567 2002, p. 91]. We describe two designs.

568 In the first design, the data type used for the abstract representation of the notional machine  
569 ( $A_{NM}$ ) is `ExpTutorDiagram`, used in Section 2.2. We represent a program in `TYPEDARITH` with the  
570 type  $Term_{TyArith}$  and the operation we focus on ( $f_{PL}$ ) is  $typeof :: Term_{TyArith} \rightarrow Maybe\ Type_{TyArith}$ ,  
571 a function that gives the type of a term (for simplicity we use a *Maybe* here to capture the cases  
572 where a term is not well-typed).

573     We then have two abstraction functions:

574      $\alpha_{Term} :: Term_{TyArith} \rightarrow ExpTutorDiagram$

575      $\alpha_{Type} :: Type_{TyArith} \rightarrow Type_{ExpTutor}$

576     The implementation of  $f_{TyArith}$ , the notional machine operation ( $f_{NM}$ ) that produces a notional  
577 machine-level representation of maybe the type of a term, is analogous to what is shown in Figure 10  
578 because (1) the abstraction function ( $\alpha_{Term}$ ) has a left inverse ( $\alpha_{Term}^\circ$ ) and (2)  $f_{PL}$  produces a *Maybe*.

579      $f_{TyArith} :: ExpTutorDiagram \rightarrow Maybe\ Type_{ExpTutor}$

580      $f_{TyArith} = fmap \alpha_{Type} \circ typeof \circ_M \alpha_{Term}^\circ$

581     As is, a student may benefit from the notional machine's representation of the program's abstract  
582 syntax tree and that may be helpful to reason about typing but the notional machine doesn't expose  
583 to the student the inner workings of the process of typing a term.

584     <sup>8</sup>The syntax and typing rules for `TYPEDARITH` are reproduced in Appendix A.2.  
585

The second design, represented in the diagram in Figure 11, tackles this issue by enriching the notional machine in a way that allows it to go step-by-step through the typing algorithm. The idea is that  $f_{NM}$  now doesn't produce a type but gradually labels each subtree with its type as part of the process of typing a term. For this,  $A_{NM}$  here is  $TyExpTreeDiagram$ , which differs from  $ExpTreeDiagram$  by adding to each node a possible type label. We still want to write  $f_{NM}$  in terms of  $f_{PL}$ . The key insight that enables this is to change  $f_{PL}$  from  $typeof$  to  $typeof1$ . The difference between  $typeof$  and  $typeof1$  is akin to the difference between big-step and small-step semantics:  $typeof1$  applies a single typing rule at a time. As a result, we have to augment our representation of a program by bundling each term with a possible type (captured in type  $TyTerm_{TyArith}$ ). The abstraction function and its left inverse are updated accordingly. The resulting notional machine is depicted in Figure 12.

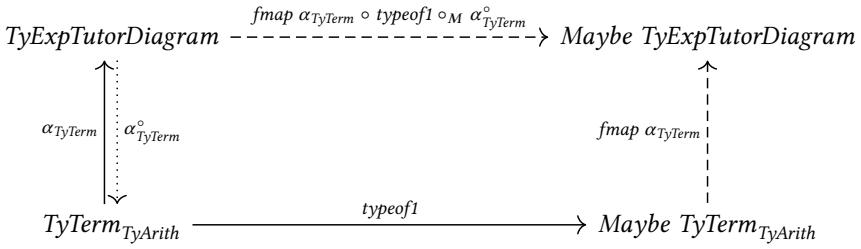


Fig. 11. Instantiation of the commutative diagram in Figure 3 for a notional machine focused on type-checking programs in TYPEDARITH. The notional machine exposes the inner workings of the typing algorithm.

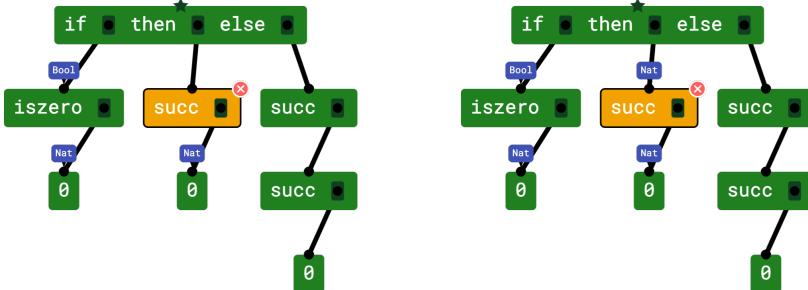


Fig. 12. One step in the notional machine TYPEDEXPTUTORDIAGRAM as it types the term if iszero 0 then succ 0 else succ succ 0 in the language TYPEDARITH.

Interestingly, given an expression  $e$ , once we label all nodes in the ExpressionTutor diagram of  $e$  with their types, the depiction of the resulting diagram is similar to the typing derivation tree of  $e$ .

Note that types are themselves trees but here we're representing them in a simplified form as textual labels because the primary goal of ExpressionTutor is to represent the structure of terms, not the structure of types.

### 3 ANALYZING EXISTING NOTIONAL MACHINES

So far we have seen how we can design sound notional machines by constructing  $f_{NM}$  using  $f_{PL}$  and functions that convert between  $A_{PL}$  and  $A_{NM}$ . Now we will analyze existing notional machines,

638 using their informal description to construct all components of the commutative diagram. In  
 639 particular, here, we have a description of  $f_{NM}$  entirely in terms of  $A_{NM}$ . We then use property-based  
 640 testing, using the soundness condition as a property, to uncover an unsoundness in the notional  
 641 machine and suggest improvements that eliminate the unsoundness.  
 642

643

644

645

646

647

648

### 3.1 Notional Machines for Data Structures

649

650 Although most notional machines are focused on the semantics of programming language constructs,  
 651 some notional machines are instead focused on data structures. One such notional machine, which  
 652 we model in this section, is the “List as Stack of Boxes” (Figure 13), described by [Du Boulay and  
 653 O’Shea 1976] and included in the dataset of notional machines analyzed by [Fincher et al. 2020].  
 654

655 Let’s see what is conceptually different between approaching a notional machine focused  
 656 on a data structure. When we presented notional machines focused on the dynamic seman-  
 657 tics of a language (EXPTREE, EXP TREE, TA-  
 658 PLMEMORYDIAGRAM), the type  $A_{PL}$  represented  
 659 a program in the language under focus (and in  
 660 the case of TAPL MEMORYDIAGRAM, addi-  
 661 tional information needed to evaluate the pro-  
 662 gram) and the function  $f_{PL}$  performed an evalua-  
 663 tion step. In the case of TYPED EXP TUTOR DIA-  
 664 GRAM, which was focused on type-checking  
 665 (the static semantics of TYPED ARITH),  $A_{PL}$  also  
 666 represented a program in that language and  $f_{PL}$   
 667 performed type-checking. Now we will model  
 668 a notional machine focused on a data structure  
 669 so  $A_{PL}$  represents that data structure and there  
 670 are several  $f_{PL}$  functions, one for each operation  
 671 supported by that data structure. The type  $A_{NM}$  can be seen as an abstraction of  $A_{PL}$  and it should  
 672 provide corresponding operations. The commutation of the diagram demonstrates the correctness  
 673 of this abstraction.

674 Using “List as Stack of Boxes” as an example, we can define a list as a data structure that supports  
 675 three operations (*Empty* and *Cons* to construct the list and *uncons* to deconstruct a list). In the  
 676 stack of boxes, each value is shown as a *String* (the textual representation of the value) in a box.  
 677 The insight of the stack of boxes is that we need a representation of an empty stack of boxes  
 678 that can be treated as a stack and not just the absence of boxes. For that we will use a pallet  
 679 (used to hold boxes in storage). To deconstruct a stack, we can pick up a box from the top of the  
 680 stack. When trying to pick up a box, either there is only the pallet, in which case we get nothing,  
 681 or there is a box on top of a stack, in which case we get the box and are left with the rest of the stack.  
 682

683

684

685

686

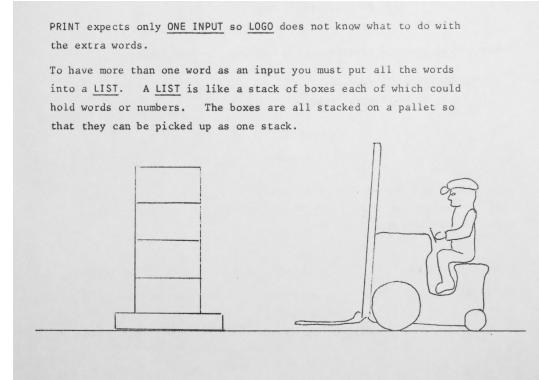


Fig. 13. The “List as Stack of Boxes” notional machine as described by Du Boulay and O’Shea [1976].

```

687
688
689  data List a where
690    Empty :: List a
691    Cons :: a → (List a) → List a
692    uncons :: List a → Maybe (a, List a)
693    uncons Empty = Nothing
694    uncons (Cons x xs) = Just (x, xs)
695
696
697  newtype Box = Box String
698  data Stack where
699    Pallet :: Stack
700    Stack :: Box → Stack → Stack
701    pickUp :: Stack → Maybe (Box, Stack)
702    pickUp Pallet = Nothing
703    pickUp (Stack box rest) = Just (box, rest)
704

```

## 3.2 A Notional Machines for Arrays

**TODO** Missing topics:

- Connect with introduction
- One way to approach the problem would be to model the subset of Java needed for the notional machine to work but Java is a fairly complex language so instead will approach this problem in a way similar to the one taken in the last section: we'll treat arrays as a data structure (a datatype) and the operations it supports.

## 3.3 Debugging a Notional Machine: The Case of Alligator Eggs

**TODO** Missing topics:

- Connect with previous section
- Describe better where property based testing comes in

Alligator Eggs<sup>9</sup> is a game conceived by Bret Victor to introduce the lambda-calculus in a playful way. It is essentially a notional machine for the untyped lambda-calculus. The game has three kinds of pieces and is guided by three rules.

**Pieces.** The pieces are *hungry alligators*, *old alligators*, and *eggs*. Old alligators are white, while hungry alligators and eggs are colored with colors other than white. The pieces are placed in a plane and their relative position with respect to each other determines their relationship. All pieces placed under an alligator are said to be guarded by that alligator. An alligator together with the pieces that may be guarded by it form a family. Families placed to the right of another family may be eaten by the guardian of the family on the left, depending on the applicability of the gameplay rules. Every egg must be guarded by an alligator with the same color (this must be a hungry alligator because eggs cannot be white).

**Rules.** There are three rules that determine the “evolution of families” over time:

**Eating rule** If there is a family guarded by a hungry alligator in the plane and there is a family or egg to its right, then the hungry alligator eats the entire family (or egg) to its right and the pieces of the eaten family are removed. The alligator that ate the pieces dies and the eggs that were guarded by this alligator and that have the same color of this alligator are hatched and are replaced by a copy of what was eaten by the alligator.

**Color rule** Before a hungry alligator *A* can eat a family *B*, if a color appears both in *A*'s protégés and in *B*, then that color is changed in one of the families to another color different from the colors already present in these families.

**Old age rule** If an old alligator is guarding only one egg or one family (which itself may be composed of multiple families), then the old alligator dies and is removed.

---

<sup>9</sup><http://worrydream.com/AlligatorEggs/>

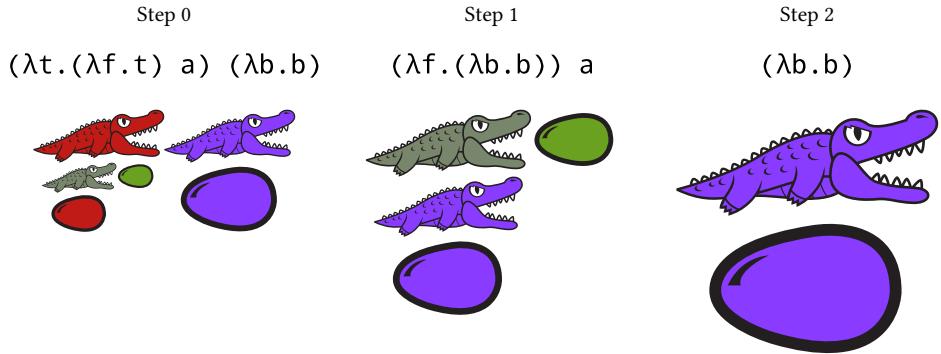


Fig. 14. Evaluation of  $(\lambda t.(\lambda f.t) a) (\lambda b.b)$  in the untyped lambda calculus (top) and ALLIGATOR notional machine (bottom).

*Relation to the Untyped Lambda-Calculus.* According to their description, the way ALLIGATOR relates to the untyped lambda-calculus is as follows: “A hungry alligator is a lambda abstraction, an old alligator is parentheses, and eggs are variables. The eating rule corresponds to beta-reduction. The color rule corresponds to (over-cautious) alpha-conversion. The old age rule says that if a pair of parentheses contains a single term, the parentheses can be removed”. Although very close, this relation is not completely accurate. We will identify the limitations and propose solutions.

*3.3.1 Illustrative Example.* Figure 14 shows a representation of the evaluation of the lambda-calculus term  $(\lambda t.(\lambda f.t) a) (\lambda b.b)$  using the ALLIGATOR notional machine. Step 0 shows the alligator families corresponding to the term. In the first step, the red alligator eats the family made of the purple alligator and the purple egg. As a result, the eaten family disappears, the red egg guarded by the red alligator hatches and is replaced by the family that was eaten, and the red alligator disappears (dies). In the second step, the grey alligator eats the green egg. As a result, the grey alligator dies and no eggs are hatched because it was not guarding any grey eggs. We are left with the purple family.

*3.3.2 Commutative Diagram.* To build a commutative diagram for ALLIGATOR, we need to build the abstract representation of the notional machine  $A_{NM}$ , which corresponds to the game pieces and the game board, the abstraction function  $\alpha :: Term_{U\lambda} \rightarrow A_{NM}$ , and an  $f_{NM}$  function, which correspond to the rules that guide the evolution of alligator families. First, we look more precisely at the game pieces and their relationship with  $Term_{U\lambda}$  to model  $A_{NM}$ .

**Eggs** An egg corresponds to a variable use and its color corresponds to the variable name.

**Hungry alligators** A hungry alligator somewhat corresponds to a lambda abstraction with its color corresponding to the name of the variable introduced by the lambda (a variable definition) and the pieces guarded by the hungry alligator corresponding to the body of the lambda abstraction. But differently from a lambda abstraction, a hungry alligator doesn't have to be guarding any pieces, which has no direct correspondence with the lambda calculus because a lambda abstraction cannot have an empty body.

**Old alligators** An old alligator somewhat corresponds to parentheses but not exactly. The lambda abstraction in the term  $(\lambda t.(\lambda f.t) a) b$  requires parentheses because conventionally the body of a lambda abstraction extends as far to the right as possible, so without the parentheses its body would be  $t a b$  instead of  $t$ . However the corresponding alligator families shown in Figure 14 don't require an old alligator. On the other hand, if we want to represent the term

785        a (b c), then we need an old alligator. Figure 15 shows an example of a term that requires  
 786        an old alligator.

787        Now let's look at the terms of the untyped lambda-calculus.  
 788        If hungry alligators are lambda abstractions and eggs are variables then what is an application? Applications are formed by  
 789        the placement of pieces on the game board. When an alligator family or egg (corresponding to a term  $t_1$ ) is placed to the left  
 790        of another family or egg (corresponding to a term  $t_2$ ), then this  
 791        corresponds to the term  $t_1$  applied to  $t_2$  (in lambda calculus  
 792        represented as  $t_1 t_2$ ).  
 793

794        Notice that because every egg must be guarded by a hungry  
 795        alligator with the same color, strictly speaking, an egg cannot  
 796        appear all by itself. That corresponds to the fact that in the  
 797        untyped lambda-calculus only lambda terms are values, so a  
 798        term cannot have unbounds variables. Textbooks, of course,  
 799        often use examples with unbound variables but these are actually metavariables that stand for an  
 800        arbitrary term. So, for convenience, we will consider that an egg by itself also forms a family.  
 801

802        We can then model an alligator family as the type *AlligatorFamily*, and a game board as simply  
 803        a list of alligator families. The result is the commutative diagram shown in Figure 16.

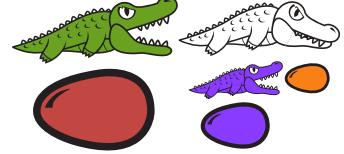


Fig. 15. Old alligator in  
 $(\lambda a.y) ((\lambda b.b) c)$

804        **data AlligatorFamily = HungryAlligator Color [AlligatorFamily]**  
 805            | **OldAlligator [AlligatorFamily]**  
 806            | **Egg Color**

807        The abstraction function  $\alpha$  relies on some function  $nameToColor :: Name \rightarrow Color$ .

$$\begin{array}{ll} 810 \quad \alpha :: Term_{U\lambda} & \rightarrow [AlligatorFamily] \\ 811 \quad \alpha \text{ (Var name)} & = [Egg (nameToColor name)] \\ 812 \quad \alpha \text{ (Lambda name } e) & = [HungryAlligator (nameToColor name) (\alpha e)] \\ 813 \quad \alpha \text{ (App } e1 e2 @ (App \_\_ \_)) & = \alpha e1 \# [OldAlligator (\alpha e2)] \\ 814 \quad \alpha \text{ (App } e1 e2) & = \alpha e1 \# \alpha e2 \end{array}$$

815  
 816        From *Proof to Property-Based Testing*. The commutativity of the diagrams presented in Chapter 2  
 817        was demonstrated using equational reasoning. Here instead, we implement the elements that consti-  
 818        tute the commutative diagram and use property-based testing to test if the diagram commutes. This  
 819        approach is less formal and it doesn't prove the notional machine correct, but it is lightweight and  
 820        potentially more attractive to users that are not familiar with equational reasoning or mechanised  
 821        proofs. We will see here that, despite its limitations, this approach can go a long way in revealing  
 822        issues with a notional machine. The idea is that a generator generates terms  $t_i :: Term_{U\lambda}$  and checks  
 823        that  $(f_{NM} \circ \alpha) t_i \equiv (\alpha \circ step) t_i$ .  
 824

825        *de Bruijn Alligators*. The first challenge is that we need to compare values of type [*AlligatorFamily*]  
 826        that were produced using  $f_{NM}$  with values produced using *step*. As we have seen, the colors in  
 827        *AlligatorFamily* correspond to variable names but the way *step* generates fresh names (which then  
 828        are turned into colors) may be different from the way  $f_{NM}$  will generate fresh colors. In fact, the  
 829        original description of *ALLIGATOR* anticipates the challenge of comparing alligator families. In  
 830        the description of possible gameplays, they clarify that to compare alligator families we need to  
 831        take into account that families with the same "color pattern" are equivalent. This can be achieved  
 832        by using a *de Bruijn representation* [de Bruijn 1972] of Alligators. We turn *AlligatorFamily* into  
 833

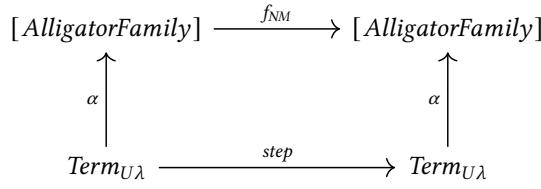


Fig. 16. First attempt at instantiating the commutative diagram in Figure 3 for the notional machine ALLIGATOR.

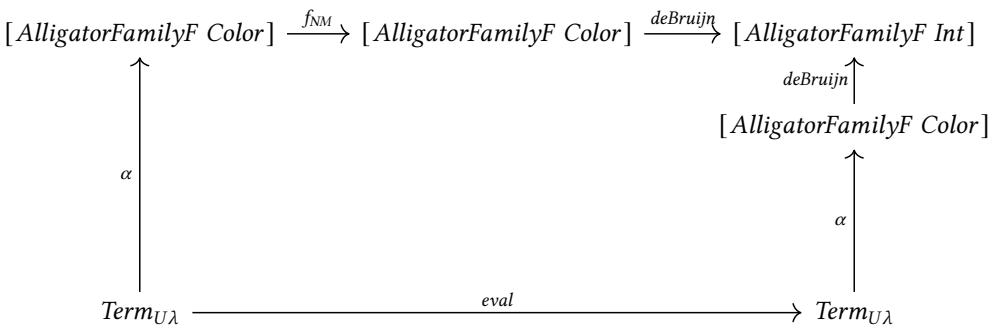
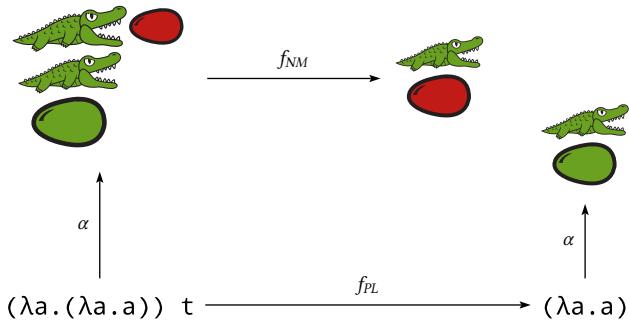


Fig. 17. Second attempt at instantiating the commutative diagram in Figure 3 for the notional machine ALLIGATOR.

AlligatorFamilyF Color and before comparing families we transform them into AlligatorFamilyF Int following the de Bruijn indexing scheme. The commutative diagram we are moving towards is shown in Figure 17.

*Evaluation Strategy.* With this setup in place, the next step is to implement  $f_{NM}$  in terms of the game rules. The eating rule (together with the color rule) somewhat corresponds to beta-reduction but under what evaluation strategy? The choice of evaluation strategy turns out to affect not only the eating rule but also the old age rule. According to the original description, any hungry alligator that has something to eat can eat and one of the original examples shows a hungry alligator eating an egg even when they are under another hungry alligator. That would correspond to a *full beta-reduction* evaluation strategy but we will stick to a call-by-value lambda-calculus interpreter so we will adapt the rules accordingly. The old age rule has to be augmented to trigger the evolution of an old alligator family that follows a topmost leftmost hungry alligator and families under a topmost leftmost old alligator. The eating rule should be triggered only for the topmost leftmost hungry alligator, unless it is followed by an old alligator (in which case the augmented old age rule applies). The color rule plays an important role in the correct behavior of the eating rule as a correspondence to beta-reduction. That's because indeed "the color rule corresponds to (over-cautious) alpha-conversion", so it is responsible for avoiding variable capture.

With all the rules implemented, we can define a function *evolve* that applies them in sequence. We will then use *evolve* in the definition of  $f_{NM}$ . One application of *evolve* corresponds to one step in the notional machine layer but that step doesn't correspond to a step in the programming language layer. For example, The main action of the old age rule (to remove old alligators) doesn't have a correspondence in the reduction of terms in UNTYPEDLAMBDA. In terms of simulation theory, in this

Fig. 18. Unsoundness: *bound* occurrences of a variable should not be substituted.

case the simulation of the programming language by the notional machine is not lock-step. To adapt our property-based testing approach, instead of making  $f_{PL}$  equal to  $step$ , we will simply reduce the term all the way to a value (leading to the use of *eval* as  $f_{PL}$  in Figure 17) and correspondingly define  $f_{NM}$  to be the successive applications of *evolve* until we reach a fixpoint.

**3.3.3 Problem: Substitution of Bound Variables.** Now we have all the building blocks of the commutative diagram. We can put them together by running the property-based tests to try to uncover issues in the diagram and indeed we do. According to the eating rule, after eating, a hungry alligator dies and if it was guarding any eggs of the same color, each of those eggs hatches into what was eaten. So the family corresponding to  $(\lambda a. (\lambda a. a)) t$ , for example, would evolve to the family corresponding to  $\lambda a. t$  instead of  $\lambda a. a$ , as shown in Figure 18. This issue corresponds to a well-known pitfall in substitution: we cannot substitute *bound* occurrences of a variable, only the ones that are *free*.

The issue can be solved in one of the following ways:

- (1) Refining the description of the eating rule, changing “if she was guarding any eggs of the same color, each of those eggs hatches into what she ate” into “if she was guarding any eggs of the same color **that are not guarded by a closer alligator with the same color**, each of those eggs hatches into what she ate”;
- (2) Restricting all colors of hungry alligators in a family to be distinct;
- (3) Defining colors to correspond to de Bruijn indices instead of names. This means that not only colors wouldn’t be repeated in the same family but also that every family would use the same “color scheme” for structurally equivalent terms.

## 4 ABSTRACT VS. CONCRETE SYNTAX OF NOTIONAL MACHINES

The ALLIGATOR notional machine we have seen uses concrete images to represent alligators and eggs. But there is a distinction between the data structure that represents the notional machine and how it is visualized. Those differences are akin to the difference between the concrete and the abstract syntax of a language. Like a programming language, a notional machine has a concrete and an abstract syntax as well. We also refer to those as concrete and abstract representations of a notional machine. Notice that the concrete representation of a notional machine may not only be a diagram or image that can be depicted on paper but it could also be made of physical objects or enacted with students. In fact, many notional machines are ludic in nature or are built around a metaphor, so the concrete representation of a notional machine is very important.

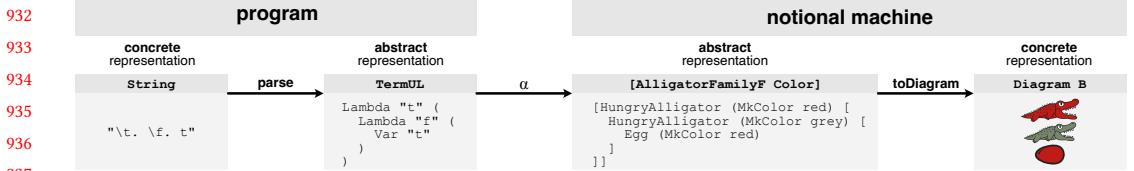


Fig. 20. Both program and notional machine have abstract and concrete representations.

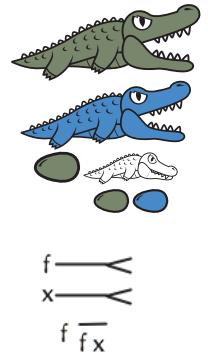


Fig. 19. Multiple concrete representations.

Figure 20 shows the different layers at play here. On the programming language side, the *parse* function converts from the concrete to the abstract syntax of the language. The function  $\alpha$  maps from language constructs to notional machine constructs. On the notional machine side, the function *toDiagram* maps from the abstract representation of the notional machine to its concrete representation, e.g., in the form of an actual diagram. In the case of **Alligator**, we use the *diagrams* library [Yates and Yorgey 2015; Yorgey 2012] to construct the concrete representation so *toDiagram* produces a value of type *Diagram B*, where the type parameter *B* (for Backend) determines the output format of the diagram (e.g. SVG).

By decoupling the abstract from the concrete representation, a notional machine can have multiple concrete representations. Alligator Eggs, for example, also describes another concrete syntax that it calls “Schematic Form”. This concrete representation is suitable for working with the notional machine using pencil and paper. Figure 19 shows the Church numeral two (the term  $\lambda f. \lambda x. f (f x)$ ), represented using both concrete representations. In the schematic representation, colors are presented with variable names. An alligator is drawn as a line ending with a < for a mouth, and is preceded by a variable name corresponding to its color. An old alligator is drawn with a line without a mouth. An egg is drawn just with the variable name corresponding to its color.

## 5 EVALUATION

The notional machines we presented so far, are meant not only to exemplify how to use our framework to reason about notional machines but also were chosen to be representative of the design space of notional machines used in practice. To characterize this design space we analyzed the notional machines in the dataset captured by Fincher et al. [2020] and classified them according to three dimensions. For each dimension, we present the values (the categories) of the dimension, the count and percentage of notional machines that fall into each value, and whether at least one of the notional machines we analyzed falls into each value.

The first dimension is the form of the notional machine (Table 2), which can be metaphorical (primarily inspired by and represented with real world objects) or diagrammatic (represented with diagrams). The distinction between the two is not always clear-cut, given that a notional machine may mix both forms and some real world objects can be represented diagrammatically. Nevertheless this distinction is useful because metaphorical notional machines may be more difficult to be made sound, given the existing degrees of freedom and constraints of the real world objects they are inspired by.

The second dimension is the broad focus of the notional machine (Table 3). We say *broad* focus because, although most notional machines in the dataset focused on the runtime semantics of a

<b>Form</b>	<b>Count</b>	<b>Percentage</b>	<b>Covered</b>
Metaphorical	19	51.35%	yes
Diagrammatic	17	45.95%	yes
Both	1	2.70%	-
<b>Total</b>	<b>37</b>	<b>100.00%</b>	<b>-</b>

Table 2. Notional machines in the dataset published by Fincher et al. [2020] classified according to their form.

<b>Broad Focus</b>	<b>Count</b>	<b>Percentage</b>	<b>Covered</b>
Evaluation	30	70%	yes
Data Structure	3	xx%	yes
Type-checking	2	xx%	yes
Parsing	2	xx%	yes
??	2	xx%	-
Logic Gates	1	xx%	no
<b>Total</b>	<b>37</b>	<b>100.00%</b>	<b>-</b>

Table 3. Notional machines in the dataset published by Fincher et al. [2020] classified according to their broad focus.

<b>Focused Construct</b>	<b>Count</b>	<b>Percentage</b>	<b>Covered</b>
Memory	7	23.33%	yes
Variables	4	13.33%	yes
Arrays	4	13.33%	yes
Functions	3	10.00%	yes
Method Calls	2	6.67%	no
Control Flow	2	6.67%	no
Call Stack	2	6.67%	no
Tracing	1	3.33%	no
String literals	1	3.33%	no
Procedure	1	3.33%	no
Objects	1	3.33%	no
Instructions	1	3.33%	no
Expression	1	3.33%	yes
<b>Total</b>	<b>30</b>	<b>70.00%</b>	<b>-</b>

Table 4. Notional machines in the dataset published by Fincher et al. [2020] classified according to their narrow focus.

programming language construct (or set of constructs), we wanted to distinguish these notional machines from others that focus on other aspects of programs, such as type-checking or parsing.

The third dimension breaks down the notion of machines that focus on evaluation into the specific constructs they focus on (Table 4).

We then classified the notion of machines shown in Section 2 and Section 3 according to these dimensions.

<b>Notional Machine</b>	<b>Form</b>	<b>Broad Focus</b>	<b>Focused Construct</b>
Array a Row of Parkings Spaces	Analogy	Evaluation	Arrays
List a Stack of Boxes	Analogy	Data Structure	List
ExpTree	Representation	Evaluation	-
ExpTutorDiagram	Representation	Evaluation	-
TAPLMemoryDiagram	Representation	Evaluation	References
TypedExpTutorDiagram	Representation	Type-checking	-
AlligatorEggs	Analogy	Evaluation	-

Table 5. Notional machines analyzed in this paper classified according to dare form, broad focus, and narrow focus.

## 5.1 Expressing Complex Language Semantics

Although the examples presented here use only small languages, the same approach can be used for larger languages. We see the dynamic and static conceptual models to reason about Ownership Types in Rust [Crichton et al. 2023] as such an example. The authors present two models: a dynamic and a static. In our framework, each model can be seen as a notional machine. The authors divide each model into a formal model (about which formal statements can be made), which would correspond to the abstract representation of the notional machine, and an informal model, which would correspond to the concrete representation of the notional machine. In the dynamic model, our PL layer would be the Rust language and our NM layer would be Miri. In the static model, our PL layer would be the Polonius model of borrow checking and our NM layer would be the *permissions model* of borrow checking (introduced by the authors).

## 6 RELATED WORK

The term notional machine was coined by Du Boulay [1986] to refer to “the idealised model of the computer implied by the constructs of the programming language”. Therefore it restricts the use of notional machines as means to help understand the runtime behavior of a program or the dynamic semantics of a language. Fincher et al. [2020], a working group that included du Boulay, has a thorough literature review and discussion of the term notional machine and established a broader definition as “pedagogic devices to assist the understanding of some aspect of programs or programming”, which includes uses of notional machines as means to help understand the static semantics of a language. The “Expressions as Trees” notional machine [Fincher et al. 2020], for example, is used for both. We adopt here this broader view of notional machine.

Another related and important term is conceptual model. Fincher et al. [2020] states that a “notional machine is effectively a special kind of conceptual model.” That’s important because conceptual models, such as the Substitution Model and the Environment Model used in the SICP book (Structure and Interpretation of Computer Programs) [Abelson et al. 1996], and SICPJS [Abelson and Sussman 2022], for example, are well understood to be essential for reasoning about programs. Notional machines that are sound are indeed conceptual models. The concrete representation of the notional machine can be thought of as embodying the visualization of this conceptual model.

The idea of simulating or representing a program by means of another program is an old one, and was first studied in detail in the 1970s by Milner [1971] and Hoare [1972]. Many of our notional machines illustrate a reduction, stepping, or evaluation ‘aspect’ of a programming language. Wadler et al. [2020] descriptions of how to relate such reduction systems with simulation, lock-step simulation, and bisimulation. The commutative diagram describing the desired property

of a notional machine appears in many places in the literature, and is a basic concept in Category Theory. Closer to our application is its use as ‘promotion condition’ [Bird 1984; Wang et al. 2013].

Where computing education researchers capture program behavior through notional machines, programming language researchers instead use semantics [Krishnamurthi and Fisler 2019]. Our work can be seen as a rather standard approach to show the correctness of one kind of semantics of (part of) a programming language, most often the operational semantics, with respect to another semantics, often a reduction semantics. An example of such an approach has been described by Clements et al. [2001], whose Elaboration Theorem describes a property that is very similar to our soundness requirement. The lack of a formal approach to showing the soundness of notional machines is also noted by Pollock et al. [2019], who develop a formal approach to specifying correct program state visualization tools, based on an executable semantics of the programming language formulated in the K framework [Roşu and Ţerbănuţă 2010]. In this paper we study a broader collection of notional machines than just program state visualization tools, and we apply our approach to study the soundness of notional machines.

Computing educators practitioners use a diverse set of notional machines [Fincher et al. 2020]. Some notional machines form the basis of automated tools. The BlueJ IDE, which features prominently in an introductory programming textbook [Kölling and Barnes 2017], includes a graphical user interface to visualize objects, invoke methods, and inspect object state. PythonTutor [Guo 2013], an embeddable web-based program visualization system, is used by hundreds of thousands of users to visualize the execution of code written in Python, Java, JavaScript, and other programming languages. UUhistle [Sorva and Sirkiä 2010], a “visual program simulation” system, takes a different approach: instead of visualizing program executions, it requires students to perform the execution steps in a constrained interactive environment. When developing such widely used tools, starting from a sound notional machine is essential.

Dickson et al. [2022] discuss the issues around developing and using a notional machine in class. They note, amongst others, “that a notional machine must by definition be correct, but a student’s mental model of the notional machine often is not”, and that “specifying a notional machine was more difficult than we thought it would be”. Our work can help in developing a notional machine, and pointing out flaws in it.

## 7 CONCLUSIONS

A notional machine are popular in computer science education, commonly used both by instructors in their teaching practice as well as by researchers. Despite their popularity, there has been no precise formal characterization of what should be the relationship between a notional machine and the aspect under its focus that would allow one to evaluate whether or not they are consistent with each other. We, therefore, introduced a definition of soundness for notional machines. The definition is based on simulation, a well-established notion widely used in many areas of computer science. Demonstrating soundness essentially amounts to constructing a commutative diagram relating the notional machine with the object of its focus.

Using this definition, we showed how we can (1) systematically design notional machines that are sound by construction, and (2) analyze existing notional machines to uncover inconsistencies and suggest improvements.

An important insight in the process is to distinguish between the concrete representation of the notional machine (typically visual) and its abstract representation, about which we can make formal statements. This distinction is akin to the distinction between the concrete and the abstract syntaxes of a programming language.

This work intends more generally to establish a framework to reason about notional machines, placing the research on notional machines on firmer ground. As such, it can be used to address

1128 challenges such as the design, analysis, and evaluation of notional machines, and the construction  
 1129 of automated tools based on notional machines.

1130

1131

## 1132 REFERENCES

- 1133 Harold Abelson and Gerald Jay Sussman. 2022. *Structure and Interpretation of Computer Programs: JavaScript Edition*. MIT  
 1134 Press.
- 1135 Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, (Second  
 1136 Edition)* (second edition ed.). Vol. 33.
- 1137 R. S. Bird. 1984. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on  
 1138 Programming Languages and Systems* 6, 4 (Oct. 1984), 487–504. <https://doi.org/10.1145/1780.1781>
- 1139 R. S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (April 1989), 122–126. [https://doi.org/10.  
 1140 1093/comjnl/32.2.122](https://doi.org/10.1093/comjnl/32.2.122)
- 1141 Jerome Bruner. 1960. *The Process of Education*. Harvard University Press.
- 1142 Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A  
 1143 Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation  
 1144 and Technology in Computer Science Education V. 1 (ITiCSE '21)*. Association for Computing Machinery, New York, NY,  
 1145 USA, 380–386. <https://doi.org/10.1145/3430665.3456343>
- 1146 Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (1936),  
 1147 345–363. <https://doi.org/10.2307/2371045> jstor:2371045
- 1148 Alonzo Church. 1941. *The Calculi of Lambda-Conversion*. Number 6. Princeton University Press.
- 1149 John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *Proceedings of the 10th  
 1150 European Symposium on Programming Languages and Systems (ESOP '01)*. Springer-Verlag, Berlin, Heidelberg, 320–334.
- 1151 Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types  
 1152 in Rust. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 265:1224–265:1252. <https://doi.org/10.1145/3622841>
- 1153 Andrew R. Dalton and William Krehling. 2010. Automated Construction of Memory Diagrams for Program Comprehension.  
 1154 In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*. Association for Computing Machinery,  
 1155 New York, NY, USA, 1–6. <https://doi.org/10.1145/1900008.1900040>
- 1156 N. G de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation,  
 1157 with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (Jan. 1972), 381–392.  
[https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0)
- 1158 Paul E. Dickson, Tim Richards, and Brett A. Becker. 2022. Experiences Implementing and Utilizing a Notional Machine in  
 1159 the Classroom. In *Proceedings of the 53rd ACM Technical Symposium V.1 on Computer Science Education (SIGCSE 2022)*.  
 1160 Association for Computing Machinery, New York, NY, USA, 850–856. <https://doi.org/10.1145/3478431.3499320>
- 1161 Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb.  
 1162 1986), 57–73. <https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9> arXiv:<https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9>
- 1163 Benedict Du Boulay and Tim O'Shea. 1976. *How to Work the LOGO Machine*. Technical Report 4. Department of Artificial  
 Intelligence, University of Edinburgh.
- 1164 Richard Feynman. 1985. *Surely You're Joking, Mr. Feynman!* W. W. Norton.
- 1165 Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas,  
 1166 Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines  
 1167 in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and  
 1168 Technology in Computer Science Education (ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA,  
 21–50. <https://doi.org/10.1145/3437800.3439202>
- 1169 Jeremy Gibbons. 2002. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program  
 1170 Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures*, Roland Backhouse,  
 Roy Crole, and Jeremy Gibbons (Eds.). Springer, Berlin, Heidelberg, 151–203. [https://doi.org/10.1007/3-540-47797-7\\_5](https://doi.org/10.1007/3-540-47797-7_5)
- 1171 Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of  
 1172 the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*. ACM Press, Denver, Colorado, USA, 579.  
<https://doi.org/10.1145/2445196.2445368>
- 1173 C. A. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1, 4 (Dec. 1972), 271–281. <https://doi.org/10.1007/BF00289507>

1174

1175

1176

## Sound Notional Machines

- 1177 Mark A. Holliday and David Luginbuhl. 2004. CS1 Assessment Using Memory Diagrams. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. Association for Computing Machinery, New York, NY, USA, 200–204. <https://doi.org/10.1145/971300.971373>
- 1178 Michael Kölling and David Barnes. 2017. *Objects First With Java: A Practical Introduction Using BlueJ* (6th ed.). Pearson.
- 1179 Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, Anthony V. Robins and Sally A. Fincher (Eds.). Cambridge University Press, Cambridge, 377–413. <https://doi.org/10.1017/9781108654555.014>
- 1180 Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Do Values Grow on Trees?: Expression Integrity in Functional Programming. In *Proceedings of the Seventh International Workshop on Computing Education Research - ICER '11*. ACM Press, Providence, Rhode Island, USA, 39. <https://doi.org/10.1145/2016911.2016921>
- 1181 Robin Milner. 1971. An Algebraic Definition of Simulation between Programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 481–489.
- 1182 Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.
- 1183 Josh Pollock, Jared Roesch, Doug Woos, and Zachary Tatlock. 2019. Theia: Automatically Generating Correct Program State Visualizations. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 46–56. <https://doi.org/10.1145/3358711.3361625>
- 1184 Grigore Rosu and Traian Florin Serbanută. 2010. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming* 79, 6 (Aug. 2010), 397–434. <https://doi.org/10.1016/j.jlap.2010.03.012>
- 1185 Juha Sorva, Jan Lönnberg, and Lauri Malmi. 2013. Students’ Ways of Experiencing Visual Program Simulation. *Computer Science Education* 23, 3 (Sept. 2013), 207–238. <https://doi.org/10.1080/08993408.2013.807962>
- 1186 Juha Sorva and Teemu Sirkia. 2010. UUHistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*. ACM Press, Berlin, Germany, 49–54. <https://doi.org/10.1145/1930464.1930471>
- 1187 Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda*.
- 1188 Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. 2013. Refactoring Pattern Matching. *Science of Computer Programming* 78, 11 (Nov. 2013), 2216–2242. <https://doi.org/10.1016/j.scico.2012.07.014>
- 1189 Ryan Yates and Brent A. Yorgey. 2015. Diagrams: A Functional EDSL for Vector Graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design (FARM 2015)*. Association for Computing Machinery, New York, NY, USA, 4–5. <https://doi.org/10.1145/2808083.2808085>
- 1190 Brent A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). *ACM SIGPLAN Notices* 47, 12 (Sept. 2012), 105–116. <https://doi.org/10.1145/2430532.2364520>
- 1191
- 1192
- 1193
- 1194
- 1195
- 1196
- 1197
- 1198
- 1199
- 1200
- 1201
- 1202
- 1203
- 1204
- 1205
- 1206
- 1207
- 1208
- 1209
- 1210
- 1211
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225

1226 **A PROGRAMMING LANGUAGE DEFINITIONS**

1227 The programming languages used throughout the paper mostly follows the presentation in the  
 1228 book Types and Programming Languages by Pierce [2002] with minor changes. In particular, italics  
 1229 are used for metavariables and the axioms in the reduction (evaluation) rules and typing rules are  
 1230 shown with explicitly empty premises.  
 1231

1232 **A.1 UntypedLambda**

1233 Figure 21 shows the syntax and evaluation rules for the untyped lambda calculus by Church [1936,  
 1234 1941], that we have referred to as UNTYPEDLAMBDA. This language is used in Sections 2.1, 2.2,  
 1235 and 3.3 .  
 1236

1237      Syntax	1238	1239      Evaluation
		$\frac{}{(\lambda x.t_1)v_2 \longrightarrow [x \mapsto v_2]t_1} \text{E-APPABS}$
$t ::=$ $x$ variable $\lambda x.t$ abstraction $t t$ application		$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \text{E-APP1}$ $\frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2} \text{E-APP2}$
$v ::=$ $\lambda x.t$		

1247 Fig. 21. The untyped lambda calculus (UNTYPEDLAMBDA).

1248 **A.2 TypedArith**

1249 We define here the language TYPEDARITH, used in Section 2.4. Figure 22 shows its syntax and  
 1250 evaluation (reduction) rules and Figure 23 shows its typing rules. The appeal of using this language  
 1251 to present a notional machine focused on the types is its simplicity. Terms don't require type  
 1252 annotations and the typing rules don't require a type environment. In fact, Pierce uses it as the  
 1253 simplest example of a typed language when introducing type safety.  
 1254

1255 **A.3 TypedLambdaRef**

1256 In Section 2.3, we showed the language TYPEDLAMBDAREF, used to design a notional machine  
 1257 that focuses on references. This language is composed of the simply-typed lambda calculus, the  
 1258 TYPEDARITH language, tuples, the Unit type, sequencing, and references. Our goal is again simplicity  
 1259 and this is the simplest language we need for the examples in the book that use the diagram. Figure 24  
 1260 shows its syntax and evaluation (reduction) rules. We show only the reduction rules for sequencing,  
 1261 references, and tuples because the rules for the rest of the language are similar to what we showed  
 1262 before, except for the store then needs to be threaded through all the rules. Although that is a typed  
 1263 language, we don't present its typing rules because the notional machine in Section 2.3 is focused  
 1264 only on its runtime behavior and not its types.  
 1265

1266

1267

1268

1269

1270

1271

1272

1273

1274

## Sound Notional Machines

1275	Syntax	Evaluation
1276		
1277		
1278		
1279		$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2} \text{E-IFTRUE}$
1280		
1281		$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3} \text{E-IFFALSE}$
1282	$t ::=$ terms:	$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \text{E-IF}$
1283	true                  constant true	
1284	false                constant false	
1285	if $t$ then $t$ else $t$ conditional	
1286	0                    constant zero	
1287	succ $t$ successor	
1288	pred $t$ predecessor	
	iszero $t$ zero test	
1289	$v ::=$ values:	
1290	true	$\frac{}{\text{succ } t_1 \longrightarrow \text{succ } t'_1} \text{E-SUCC}$
1291	false	
1292	nv	
1293	$nv ::=$ numeric values:	
1294	0	$\frac{}{\text{pred } 0 \longrightarrow 0} \text{E-PREDZERO}$
1295	succ $nv$	
1296		$\frac{}{\text{pred } (\text{succ } nv_1) \longrightarrow nv_1} \text{E-PREDSUCC}$
1297		
1298		$\frac{t_1 \longrightarrow t'_1}{\text{pred } t_1 \longrightarrow \text{pred } t'_1} \text{E-PRED}$
1299		
1300		$\frac{}{\text{iszero } 0 \longrightarrow \text{true}} \text{E-ISZEROZERO}$
1301		
1302	Fig. 22. Syntax and reduction rules of the TYPEDARITH language.	
1303		
1304		
1305		
1306		
1307		
1308		
1309		
1310		
1311		
1312		
1313		
1314		
1315		
1316		
1317		
1318		
1319		
1320		
1321		
1322		
1323		

1324	Syntax	Typing rules
1325		
1326		$\frac{}{\text{true} : \text{Bool}} \text{T-TRUE}$
1327		$\frac{}{\text{false} : \text{Bool}} \text{T-FALSE}$
1328		$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{T-IF}$
1329		
1330		
1331		
1332	$T ::=$ types: Bool           type of booleans   Nat          type of natural numbers	$\frac{}{0 : \text{Nat}} \text{T-ZERO}$
1333		$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \text{T-SUCC}$
1334		$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \text{T-PRED}$
1335		
1336		
1337		
1338		
1339		$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \text{T-ISZERO}$
1340		
1341		
1342	Fig. 23. Syntax of types and typing rules of the <code>TYPEDARITH</code> language.	
1343		
1344		
1345		
1346		
1347		
1348		
1349		
1350		
1351		
1352		
1353		
1354		
1355		
1356		
1357		
1358		
1359		
1360		
1361		
1362		
1363		
1364		
1365		
1366		
1367		
1368		
1369		
1370		
1371		
1372		

## Sound Notional Machines

1373	Syntax	Evaluation
1374	$t ::=$ terms:	
1375	$x$ variable	
1376	$\lambda x : T. t$ abstraction	
1377	$t t$ application	
1378	$\mid \text{true}$ boolean true	
1379	$\mid \text{false}$ boolean false	
1380	$\mid 0$ zero	
1381	$\mid \text{succ } t$ successor	
1382	$\mid \text{pred } t$ predecessor	
1383	$\mid \text{iszero } t$ zero test	
1384	$\mid \text{unit}$ unit constant	
1385	$\mid t; t$ sequence	
1386	$\mid \text{ref } t$ reference creation	
1387	$\mid !t$ dereference	
1388	$\mid t := t$ assignment	
1389	$\mid l$ location	
1390	$\mid \{t_i\}_{i \in 1..n}$ tuple	
1391	$\mid t.i$ projection	
1392	$v ::=$ values:	
1393	$\lambda x : T. t$	
1394	$\mid \text{true}$	
1395	$\mid \text{false}$	
1396	$\mid 0$	
1397	$\mid \text{succ } v$	
1398	$\mid \text{unit}$	
1399	$\mid l$	
1400	$\mid \{v_i\}_{i \in 1..n}$	
1401	$T ::=$ types:	
1402	$T \rightarrow T$ function type	
1403	$\mid \text{Bool}$ boolean type	
1404	$\mid \text{Nat}$ natural number type	
1405	$\mid \text{Unit}$ unit type	
1406	$\mid \text{Ref } T$ reference type	
1407	$\mid \{T_i\}_{i \in 1..n}$ tuple type	
1408	$\mu ::=$ store:	
1409	$\emptyset$ empty store	
1410	$\mid \mu, l \mapsto v$ location binding	
1411		
1412		Fig. 24. <b>TYPEDLAMBDAREF</b> : Syntax and Evaluation
1413		
1414		
1415		
1416		
1417		
1418		
1419		
1420		
1421		
		$\frac{t_1   \mu \longrightarrow t'_1   \mu'}{t_1; t_2   \mu \longrightarrow t'_1; t_2   \mu'} \text{E-SEQ}$
		$\frac{}{\text{unit}; t_2   \mu \longrightarrow t_2   \mu} \text{E-SEQNEXT}$
		$\frac{l \notin \text{dom}(\mu)}{\text{ref } v_1   \mu \longrightarrow l   (\mu, l \mapsto v_1)} \text{E-REFV}$
		$\frac{t_1   \mu \longrightarrow t'_1   \mu'}{\text{ref } t_1   \mu \longrightarrow \text{ref } t'_1   \mu'} \text{E-REF}$
		$\frac{\mu(l) = v}{!l   \mu \longrightarrow v   \mu} \text{E-DEREFLOC}$
		$\frac{t_1   \mu \longrightarrow t'_1   \mu'}{!t_1   \mu \longrightarrow !t'_1   \mu'} \text{E-DEREF}$
		$\frac{l := v_2   \mu \longrightarrow \text{unit}   [l \mapsto v_2]   \mu}{l := v_2   \mu \longrightarrow \text{unit}   [l \mapsto v_2]   \mu} \text{E-ASSIGN}$
		$\frac{t_1   \mu \longrightarrow t'_1   \mu'}{t_1 := t_2   \mu \longrightarrow t'_1 := t_2   \mu'} \text{E-ASSIGN1}$
		$\frac{t_2   \mu \longrightarrow t'_2   \mu'}{v_1 := t_2   \mu \longrightarrow v_1 := t'_2   \mu'} \text{E-ASSIGN2}$
		$\frac{\{v_i\}_{i \in 1..n}.j   \mu \longrightarrow v_j   \mu}{\{v_i\}_{i \in 1..n}.j   \mu \longrightarrow v_j   \mu} \text{E-PROJTUPLE}$
		$\frac{t_1   \mu \longrightarrow t'_1   \mu'}{t_1.i   \mu \longrightarrow t'_1.i   \mu'} \text{E-PROJ}$
		$\frac{t_j   \mu \longrightarrow t'_j   \mu'}{\{v_i\}_{i \in 1..j-1}, t_j, t_k^{k \in j+1..n}   \mu \longrightarrow \{v_i\}_{i \in 1..j-1}, t'_j, t_k^{k \in j+1..n}   \mu'} \text{E-TUPLE}$