Sound Notional Machines

ANONYMOUS AUTHOR(S)

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

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

Additional Key Words and Phrases: notional machines, programming education, equational reasoning, bisimulation

1 INTRODUCTION

Learning to program involves learning how to express a program in a programming language, but also learning what the semantics of such a program is. For novices, the semantics of a program is often not obviously apparent from the program itself. Instructors then often use a *notional machine* [Fincher et al. 2020] to help teach some particular aspect of programs and programming languages, and also to assess students' understanding of said aspect. This aspect is the notional machine's focus. For example, showing expressions as trees (the ExpTree notional machine), as depicted in Figure 1, brings out the internal structure of lambda-calculus expressions [Marceau et al. 2011] and can help to explain the step-by-step evaluation of such expressions. Notional machines are used widely in computer science education; Fincher et al. [2020] interviewed computer science teachers to build up a dataset of 37 notional machines[1].

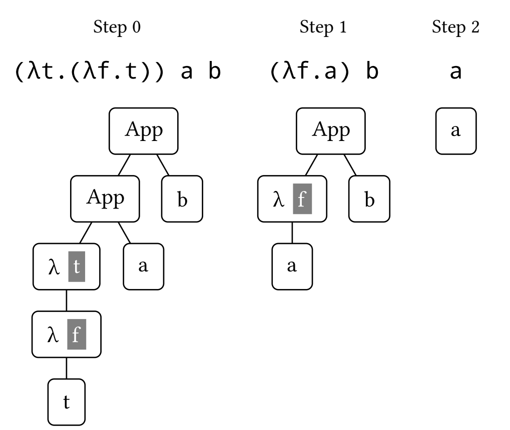


Fig. 1. Evaluation of the term `(λt.(λf.t)) a b` shown with the notional machine ExpTree.

1.1 Unsound Notional Machines

Given the extensive use of notional machines, and their intended use as devices to help students when learning, it is important to look at their quality. To begin with, one should make sure that the notional machine is *sound*: it is faithful to the aspect of programs it is meant to focus on. Anecdotal

[1]https://notionalmachines.github.io/notional-machines.html - The website lists 57 items but some of them are part of what they call a notional machine sequence, which we consider a single notional machine.

evidence of using unsound representations in education goes back a long way. In 1960, education pioneer Jerome Bruner wrote that "the task of teaching a subject to a child at any particular age is one of representing the structure of that subject in terms of the child's way of viewing things", that this "task can be thought of as one of translation", but that ideas have to be "represented *honestly* [. . .] in the thought forms of children" [Bruner 1960]. Bruner's use of the term "honestly" can be seen as a call for soundness of such representations. Decades later, Richard Feynman eloquently stated [Feynman 1985], after reviewing "seventeen feet" of new mathematics schoolbooks for the California State Curriculum Commission:

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

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 **`int`**s, for example, then when we instantiate an array, all its slots contain `0`, 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 out that, "the effectiveness of the analogy depends on [...] how well that models the semantics of the programming language."

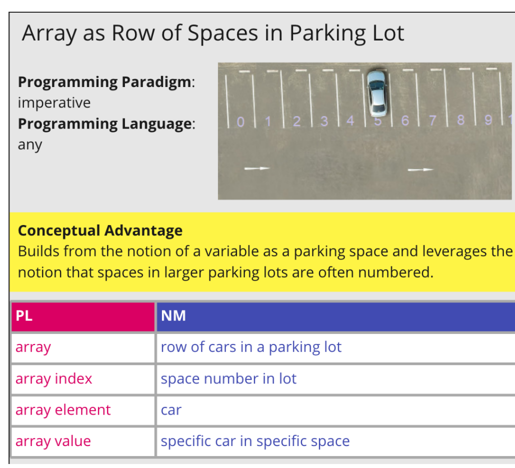


Fig. 2. The "Array as Row of Spaces in Parking Lot" notional machine captured by Fincher et al. [2020].

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 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. Cousot and Cousot [1977] use a similar commutative diagram to simulate concrete computations in their abstract interpretation framework. 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.

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[2]. The

[2]The artifact containing a superset of the examples in this paper is at Anonymous.

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	Type-checking

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

$$\begin{tikzcd} A_{NM} \arrow[r, "f_{NM}"] & B_{NM} \\ A_{PL} \arrow[u, "\alpha_A"] \arrow[r, "f_{PL}"] & B_{PL} \arrow[u, "\alpha_B"'] \end{tikzcd}$$

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

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

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[3]). 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).

The bottom layer (A_{PL}, f_{PL}, B_{PL}) represents the aspect of a programming language[4] we want to focus on. A_{PL} is an abstract representation of a program in that language. In our example, that is the

[3] The syntax and reduction rules for UntypedLambda are reproduced in the appendix provided as supplementary material .
[4] 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.

$$\begin{tikzcd}
ExpTree \arrow[rr, dashed, "\alpha \circ step \circ \alpha^{-1}"] \arrow[d, shift left, "\alpha^{-1}"] & & ExpTree \\
Term_{U\lambda} \arrow[rr, "step"] \arrow[u, shift left, "\alpha"] & & Term_{U\lambda} \arrow[u, "\alpha"']
\end{tikzcd}$$

Fig. 4. Instantiation of the commutative diagram in Figure 3 for the notional machine ExpTree and the programming language UntypedLambda.

abstract syntax of UntypedLambda (given by the type $Term_{U\lambda}$). The function f_{PL} is an operation the notional machine is focusing on. In our example, that would be *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 $Term_{U\lambda}$.

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 *ExpTree* trivially isomorphic to $Term_{U\lambda}$ 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 α_A and α_B from the abstract representation of a program in the programming language to the abstract representation of the notional machine. α 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}:

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

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

$$f_{NM} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \circ {\alpha_A}^{-1} \circ \alpha_A \equiv \alpha_B \circ f_{PL} \tag{2}$$

Instantiating the commutative diagram for ExpTree and 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.

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

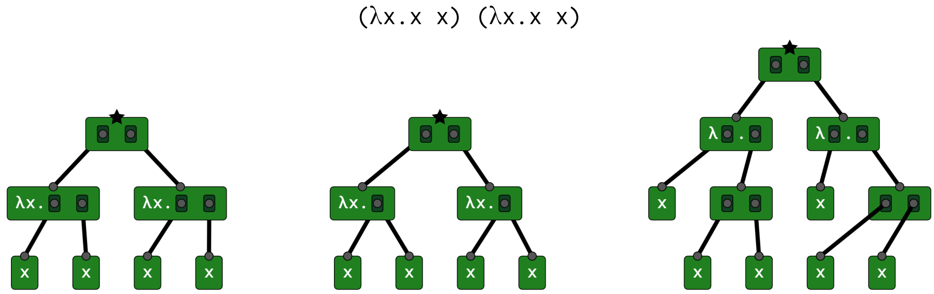


Fig. 5. The omega combinator in UntypedLambda (top) and (incorrect) representations in ExpTutorDiagram notional machine (bottom).

2.2 Monomorphic Notional Machines

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

ExpressionTutor[5] 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.

We define the notional machine ExpTutorDiagram, which models the behavior of ExpressionTutor. The fact that ExpressionTutor allows students to construct incorrect expression tree diagrams means that the abstraction function α is not bijective, as was the case of ExpTree's α. Such incorrect diagrams do not correspond to programs, thus α is deliberately not surjective.

2.2.1 Illustrative Example. Figure 5 uses ExpTutorDiagram 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 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 α_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

[5] expressiontutor.org

$$
\begin{tikzcd}
ExpTutorDiagram \arrow[rrr, dashed, "fmap\ \alpha \circ fmap\ step \circ \alpha^{\circ}"] \arrow[dr, "\alpha^{\circ}"] & & & Maybe\ ExpTutorDiagram & \\
& Maybe\ Term_{U\lambda} \arrow[rrr, dashed, "fmap\ step"] & & & Maybe\ Term_{U\lambda} \arrow[ul, dashed, "fmap\ \alpha"'] \\
Term_{U\lambda} \arrow[uu, "\alpha"] \arrow[ur, dashed, "return"] \arrow[rrr, "step"] & & & Term_{U\lambda} \arrow[uu, dashed, "fmap\ \alpha \circ return"] \arrow[ur, dashed, "return"] &
\end{tikzcd}
$$

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

operations on A_{PL}. For this we define a function $\alpha_A^{\circ} :: A_{NM} \longrightarrow Maybe\ A_{PL}$ to be a left inverse of α_A such that $\alpha_A^{\circ} \circ \alpha_A \equiv return$ (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 $Maybe\ B'_{NM}$ and the mapping α_B can be implemented in terms of a mapping $\alpha'_B :: B_{PL} \longrightarrow B'_{NM}$ like so:

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

Using the left inverse α_A° and α'_B, we define the operation on A_{NM} as follows:

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

This square commutes like so:

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

$$
\begin{aligned}
& \alpha_B \circ f_{PL} \\
\equiv\ & \{ \text{definition of } \alpha_B \} \\
& return \circ \alpha'_B \circ f_{PL} \\
\equiv\ & \{ \text{third monad law} \} \\
& fmap\ \alpha'_B \circ return \circ f_{PL}
\end{aligned}
$$

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 $Term_{U\lambda}$), with f_{PL} equal to *step*.

The construction of a mapping $\alpha :: Term_{U\lambda} \longrightarrow ExpTutorDiagram$ is straightforward because a term $t :: Term_{U\lambda}$ forms a tree and from it we can always construct a corresponding ExpressionTutor diagram d::*ExpTutorDiagram* (a graph). For each possible term in $Term_{U\lambda}$, we need to define a pattern

$$
\begin{tikzcd}
\mathit{ExpTutorDiagram} \arrow[rr, dashed, "\mathit{fmap}\ \alpha \circ \mathit{fmap}\ \mathit{step} \circ \alpha^{\circ}"] \arrow[d, dotted, shift left, "\alpha^{\circ}"] & & \mathit{Maybe}\ \mathit{ExpTutorDiagram} \\
\mathit{Term}_{U\lambda} \arrow[u, shift left, "\alpha"] \arrow[rr, "\mathit{step}"] & & \mathit{Term}_{U\lambda} \arrow[u, dashed, "\mathit{return} \circ \alpha"']
\end{tikzcd}
$$

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

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 $\alpha^{\circ} :: \mathit{ExpTutorDiagram} \rightarrow \mathit{Maybe}\ \mathit{Term}_{U\lambda}$ 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 $\mathit{Term}_{U\lambda}$ node, besides making sure that they are connected in a way that indeed corresponds to a valid $\mathit{Term}_{U\lambda}$ tree.

In the next section, we construct another commutative diagram where f_{NM} is defined using f_{PL} and a left inverse mapping α°_{A}. 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 α°_{A} is of type $A_{NM} \rightarrow \mathit{Maybe}\ A_{PL}$ and not $A_{NM} \rightarrow A_{PL}$) and omit the path via $\mathit{Maybe}\ A_{PL}$ 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 Kreahling 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.

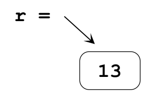
2.3 A Monomorphic Notional Machine to Reason About State

A common use of notional machines is in the context of reasoning about state. In fact, 16 of the 37 notional machines in the dataset by Fincher et al. [2020] focus on either References, Variables, or Arrays (see Section 5). 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[6]). 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 pointed to by that reference. We will refer to this notation, which we will develop into a notional machine, as TAPLMemoryDiagram. In this notation, references are represented as arrows and cells are represented as rounded rectangles containing the representation of the value contained in the cell. Before designing the notional machine, we need to see the context in which this notation is used in the book.

The book first uses this notation to explain the effect of creating a reference. It shows that when we reduce the term `ref 13` we obtain a reference (a store location) to a store cell containing 13.

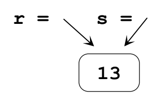
[6]The syntax and reduction rules for TypedLambdaRef are reproduced in the appendix provided as supplementary material .

The book then represents the result of binding the name `r` to such a reference with the following diagram:



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

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



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

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

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

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

(1) Bind `r` to the result of evaluating `ref 13`
(2) Bind `s` to the result of evaluating `r`
(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:

```
(λr:Ref Nat.(λs:Ref Nat.s := 82; !r) r) (ref 13)
```

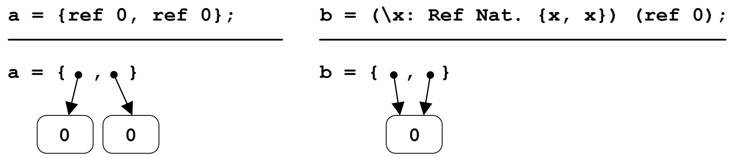


Fig. 9. TAPLMemoryDiagram for TypedLambdaRef for TAPL Exercise 13.1.1

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 9. 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 8 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

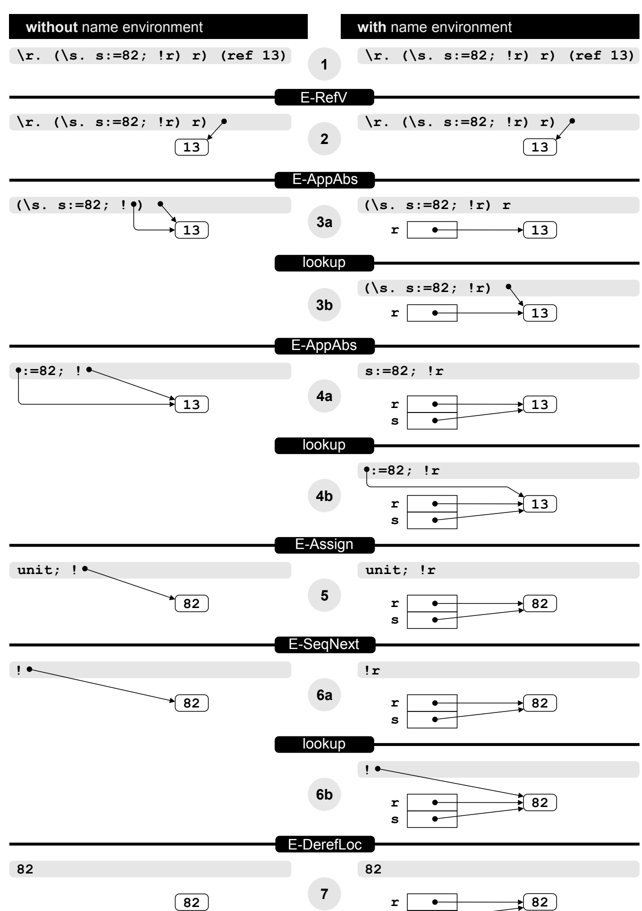


Fig. 8. Using TAPLMemoryDiagram to trace the evaluation of `(λr:Ref Nat.(λs:Ref Nat.s := 82; !r) r) (ref 13)`

$$\begin{tikzcd}
TAPLMemoryDiagram \arrow[rr, dashed, "fmap\ \alpha \circ step \circ_M \alpha^\circ"] \arrow[d, dotted, shift left, "\alpha^\circ"] & & Maybe\ TAPLMemoryDiagram \\
(Term_{U\lambda}, Env_{U\lambda}, Store_{U\lambda}) \arrow[u, shift left, "\alpha"] \arrow[rr, "step"] & & Maybe\ (Term_{U\lambda}, Env_{U\lambda}, Store_{U\lambda}) \arrow[u, dashed, "fmap\ \alpha"']
\end{tikzcd}$$

Fig. 10. Instantiation of the commutative diagram in Figure 3 for the notional machine TAPLMemoryDiagram and the programming language TypedLambdaRef.

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.

```
data TAPLMemoryDiagram l = TAPLMemoryDiagram {
  memDiaTerm :: DTerm l,
  memDiaNameEnv :: Map Name (DTerm l),
  memDiaStore :: Map (DLocation l) (DTerm l)}

data DTerm l = Leaf String
             | Branch [DTerm l]
             | TLoc (DLocation l)
```

The type *DLocation* corresponds to arrow destinations (arrow endpoints). A term is represented as a rose tree of *String*s augmented with a case for location.

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

2.4 A Monomorphic Notional Machine to Reason About Types

So far we have seen examples of commutative diagrams where f_{PL} is *step* (a function that performs a reduction step) but in principle, f_{PL} could be any operation on A_{PL} that is the focus of a given notional machine. Let's look at an example of notional machine where we do not focus on evaluating but on typing an expression. The language this notional machine focuses on is TypedArith[7], a language of typed arithmetic expression, which is the simplest typed language introduced in TAPL [Pierce 2002, p. 91]. We describe two designs.

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

We then have two abstraction functions:

$$\alpha_{Term} :: Term_{TyArith} \rightarrow ExpTutorDiagram$$
$$\alpha_{Type} :: Type_{TyArith} \rightarrow Type_{ExpTutor}$$

The implementation of $f_{TyArith}$, the notional machine operation (f_{NM}) that produces a notional machine-level representation of maybe the type of a term, is analogous to what is shown in Figure 10 because (1) the abstraction function (α_{Term}) has a left inverse (α°_{Term}) and (2) f_{PL} produces a Maybe.

$$f_{TyArith} :: ExpTutorDiagram \rightarrow Maybe\ Type_{ExpTutor}$$
$$f_{TyArith} = fmap\ \alpha_{Type} \circ typeof \circ_{M} \alpha^{\circ}_{Term}$$

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

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 does not 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.

$$\begin{tikzcd}
TyExpTutorDiagram \arrow[r, dashed, "fmap\ \alpha_{TyTerm} \circ typeof1 \circ_{M} \alpha^{\circ}_{TyTerm}"] \arrow[d, dotted, shift left, "\alpha^{\circ}_{TyTerm}"] & Maybe\ TyExpTutorDiagram \\
TyTerm_{TyArith} \arrow[u, shift left, "\alpha_{TyTerm}"] \arrow[r, "typeof1"] & Maybe\ TyTerm_{TyArith} \arrow[u, dashed, "fmap\ \alpha_{TyTerm}"']
\end{tikzcd}$$

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.

[7] The syntax and typing rules for TypedArith are reproduced in the appendix provided as supplementary material .

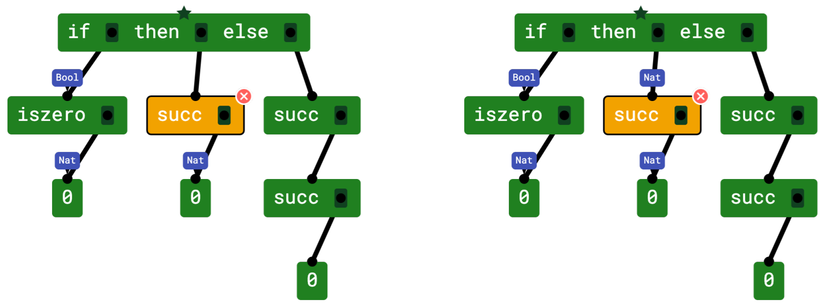


Fig. 12. One step in the notional machine TypedExpTutorDiagram as it types the term `if iszero 0 then succ 0 else succ succ 0` in the language TypedArith.

Section	Notional Machine	Programming Language	Focus
3.1	Alligator	UntypedLambda	Evaluation
3.2	ListAsStack	-	Data Structure (List)
3.3	ArrayAsParkingSpots	Java	Evaluation (Arrays)

Table 2. Notional machines, programming languages, and aspects of focus used in Section 3.

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, using their informal description to construct all components of the commutative diagram. In particular, here, we have a description of f_{NM} entirely in terms of A_{NM}. We then use property-based testing, using the soundness condition as a property, to uncover unsoundnesses and suggest improvements that eliminate them. Table 2 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.

3.1 Debugging a Notional Machine: The Case of Alligator Eggs

Alligator Eggs[8] 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

[8] http://worrydream.com/AlligatorEggs/

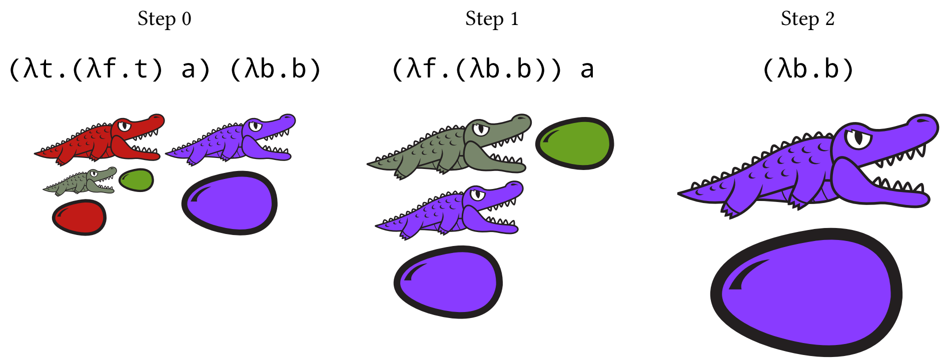


Fig. 13. Evaluation of `(λt.(λf.t) a) (λb.b)` in the untyped lambda calculus (top) and Alligator notional machine (bottom).

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 proteges 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.

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.

3.1.1 Illustrative Example. Figure 13 shows a representation of the evaluation of the lambda-calculus term `(λt.(λf.t) a) (λb.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.

3.1.2 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 $\alpha :: Term_{U\lambda} \longrightarrow A_{NM}$, 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 does not 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 `(λt.λf.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 13 don't require an old alligator. On the other hand, if we want to represent the term `a (b c)`, then we need an old alligator. Figure 14 shows an example of a term that requires an old alligator.

Now let's look at the terms of the untyped lambda-calculus. If hungry alligators are lambda abstractions and eggs are variables then what is an application? Applications are formed by 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 of another family or egg (corresponding to a term t_2), then this corresponds to the term t_1 applied to t_2 (in lambda calculus represented as $t_1\ t_2$).

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

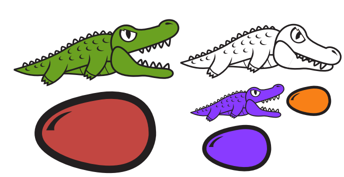


Fig. 14. Old alligator in `(λa.y) ((λb.b) c)`

We can then model an alligator family as the type *AlligatorFamily*, and a game board as simply a list of alligator families. The result is the commutative diagram shown in Figure 15.

$$
\begin{aligned}
&\textbf{data}\ \mathit{AlligatorFamily} = \mathit{HungryAlligator}\ \mathit{Color}\ [\mathit{AlligatorFamily}]\\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{OldAlligator}\ [\mathit{AlligatorFamily}]\\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{Egg}\ \mathit{Color}
\end{aligned}
$$

The abstraction function α relies on some function $\mathit{nameToColor} :: \mathit{Name} \longrightarrow \mathit{Color}$.

$$
\begin{array}{lll}
\alpha :: Term_{U\lambda} & \longrightarrow & [\mathit{AlligatorFamily}]\\
\alpha\ \ (\mathit{Var}\ \mathit{name}) & = & [\mathit{Egg}\ (\mathit{nameToColor}\ \mathit{name})]\\
\alpha\ \ (\mathit{Lambda}\ \mathit{name}\ e) & = & [\mathit{HungryAlligator}\ (\mathit{nameToColor}\ \mathit{name})\ (\alpha\ e)]\\
\alpha\ \ (\mathit{App}\ e1\ e2@(\mathit{App}\ \_\ \_)) & = & \alpha\ e1 +\!\!+ [\mathit{OldAlligator}\ (\alpha\ e2)]\\
\alpha\ \ (\mathit{App}\ e1\ e2) & = & \alpha\ e1 +\!\!+ \alpha\ e2
\end{array}
$$

$$\begin{tikzcd}
[\mathit{AlligatorFamily}] \arrow[r, "f_{NM}"] & [\mathit{AlligatorFamily}] \\
\mathit{Term}_{U\lambda} \arrow[u, "\alpha"] \arrow[r, "\mathit{step}"] & \mathit{Term}_{U\lambda} \arrow[u, "\alpha"']
\end{tikzcd}$$

Fig. 15. First attempt at instantiating the commutative diagram in Figure 3 for the notional machine ALLIGATOR.

$$\begin{tikzcd}
[\mathit{AlligatorFamilyF\ Color}] \arrow[r, "f_{NM}"] & [\mathit{AlligatorFamilyF\ Color}] \arrow[r, "\mathit{deBruijn}"] & [\mathit{AlligatorFamilyF\ Int}] \\
& & [\mathit{AlligatorFamilyF\ Color}] \arrow[u, "\mathit{deBruijn}"] \\
\mathit{Term}_{U\lambda} \arrow[uu, "\alpha"] \arrow[rr, "\mathit{eval}"] & & \mathit{Term}_{U\lambda} \arrow[u, "\alpha"]
\end{tikzcd}$$

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

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

de Bruijn Alligators. The first challenge is that we need to compare values of type $[\mathit{AlligatorFamily}]$ that were produced using f_{NM} with values produced using *step*. As we have seen, the colors in *AlligatorFamily* correspond to variable names but the way *step* generates fresh names (which then are turned into colors) may be different from the way f_{NM} will generate fresh colors. In fact, the original description of ALLIGATOR anticipates the challenge of comparing alligator families. In the description of possible gameplays, they clarify that to compare alligator families we need to take into account that families with the same "color pattern" are equivalent. This can be achieved by using a *de Bruijn representation* [Bruijn 1972] of Alligators. We turn *AlligatorFamily* into *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 16.

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

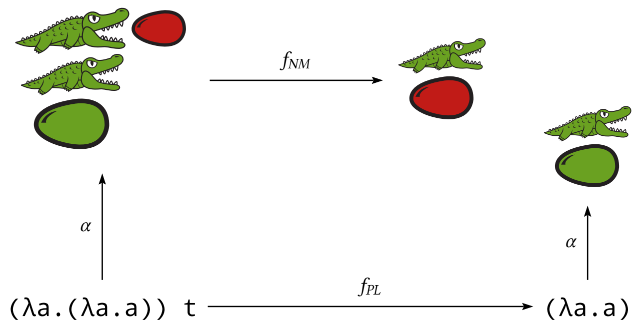


Fig. 17. Unsoundness: *bound* occurrences of a variable should not be substituted.

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 does not correspond to a step in the programming language layer. For example, The main action of the old age rule (to remove old alligators) does not have a correspondence in the reduction of terms in UntypedLambda. In terms of simulation theory, in this 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 16) and correspondingly define f_{NM} to be the successive applications of *evolve* until we reach a fixpoint.

3.1.3 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 `(λa.(λa.a)) t`, for example, would evolve to the family corresponding to `λa.t` instead of `λa.a`, as shown in Figure 17. 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.

3.2 Notional Machines for Data Structures

Although most notional machines focus on the semantics of programming language constructs, some notional machines instead focus on data structures. One such notional machine, which we model in this section, is the "List as Stack of Boxes" (Figure 18), described by Du Boulay and O'Shea [1976] and included in the dataset of notional machines analyzed by Fincher et al. [2020].

For notional machines focussing on the dynamic semantics of a language (ExpTree, ExpTree, TAPLMemoryDiagram), the type A_{PL} represents a program in the language under focus (and in the case of TAPLMemoryDiagram, additional information needed to evaluate the program) and the function f_{PL} performs an evaluation step. In the case of TypedExpTutorDiagram, which focusses on type-checking (the static semantics of TypedArith), A_{PL} also represents a program in that language and f_{PL} performed type-checking. Now we will model a notional machine focussing on a data structure, so A_{PL} represents that data structure and there are several f_{PL} functions, one for each operation supported by that data structure. The type A_{NM} can be seen as an abstraction of A_{PL} and it should provide corresponding operations. The commutation of the diagram demonstrates the correctness of this abstraction.

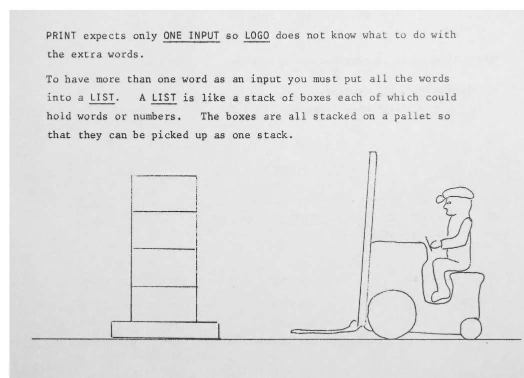


Fig. 18. The "List as Stack of Boxes" notional machine as described by Du Boulay and O'Shea [1976].

Modelling "List as Stack of Boxes" requires some adaptations to the original description:

(1) To avoid partiality of the operations typically used to access the head and tail of a list (FIRST and REST in the original description), we define a list using three operations: *Empty* and *Cons* to construct a list and $uncons :: List\ a \rightarrow Maybe\ (a, List\ a)$ to deconstruct it. That change also has a covenient representation in the notional machine.
(2) In the stack of boxes, each value is shown as a *String* in a box, which means we can only represent lists of values for which we can create a *String* representation.
(3) The original description wasn't explicit about the representation of an empty list. We need a corresponding empty stack of boxes that can be treated as a stack and not just the absence of boxes. For that we will use a pallet (used to hold boxes in storage). The original description mentions a pallet in two contexts:
   (a) "boxes are stacked on a pallet so that they can be picked up as one stack". That's an important part of the notional machine's behavior but we also need to be able to pick up a box from the top of the stack;
   (b) "The beginning of a list, the top of the stack, is marked with [and the end of the list, the pallet is marked with]". Here there's an asymmetry between the end of the list, represented with a pallet, and the beginning of the list, which has no representation in the notional machine. Our denotation of pallet is different: it is the representation of an empty list.

The result is that we can construct lists either with a pallet or by stacking a box on top of a stack, which must have eventually a pallet at the end. To deconstruct it, we can pick up a box from the top of the stack ($pickUp :: Stack \rightarrow Maybe\ (Box, Stack)$). When trying to pick up a box, either there is only the pallet, in which case we get nothing, or there is a box on top of a stack, in which case we are left with the box and the rest of the stack.

The process of formalizing the notional machine, making it precise, once again helped to identify the issues and improve the notional machine.

3.3 A Notional Machines for Arrays

With the techniques we have developed so far, we can turn again to the notional machine "Array as Row of Parking Spaces is Parking Lot", presented in Section 1.1. There we considered Java as the underlying programming language and already identified the first issue, caused by the difference between the representation of arrays of primitive types and arrays of reference types. Here, we will make the notional machine more precise and in the process resolve two problems.

One approach would be to model the subset of Java needed for the notional machine to work, but Java is a fairly complex language. Instead we start by following the approach taken in the last section: we model the PL layer as an idealized array and the operations it supports. An array supports three operations: *allocating* an array of a given type and a given length, *reading* an element at a given index, and *writing* an element to a given index.

3.3.1 Problem: reference types versus primitive types. One of the appeals of the notional machine is that we would represent a newly allocated array of objects as an empty parking lot, because it does not contain "valid" values (their slots contain *null*). The first issue is that arrays of primitive types are really different in that, when newly allocated, their slots already contain valid values. These arrays would be represented as fully booked parking lots with slots that can never be empty! Here, instead of changing the design of the notional machine, we can use this misfit in the metaphor as a learning opportunity and explicitly discuss it with students. There is still a decision to be made about how exactly the values in the array are going to be represented. We could represent a value as a car with the string representation of the value drawn on its roof, but representing both unboxed and boxed versions of an integer with the same string could be confusing. Instead, we could opt for representing values of reference types as cars of a different color, for example. Of course an array can also contain other arrays, which are themselves objects, and that complicates the picture.

3.3.2 Problem: representing multiple arrays. To represent multiple arrays, and the relationship between them, we cannot model the PL layer simply as an idealized array[9]. Instead, we use the programming language LambdaRef augmented with arrays. We first used this language in Section 2.3, where the TAPLMemoryDiagram notional machine represented all constructs of the language as well as the program's state as it ran (its memory): the store and name environment. Here we want to represent only arrays and values so the notional machine layer contains, besides the parking lot, essentially a sequence of statements equivalent to the occurrences of all terms in the program that manipulate arrays (an array allocation, array access, and assignment of a value to an array slot) as they happen when the program runs. This way we can ignore all other terms in the language and aspects of the program's memory as it runs. Besides the information present in the array-manipulating terms themselves, we need one more thing: to be able to uniquely identify each array (e.g. with its location, or address). We annotate each parking lot with the corresponding location (for example using an @ and the location identifier) which we use to identify the parking lot when we write to (or read from) its slots.

[9] The representation of lists shown in Section 3.2 is not suitable to represent lists that contain lists.

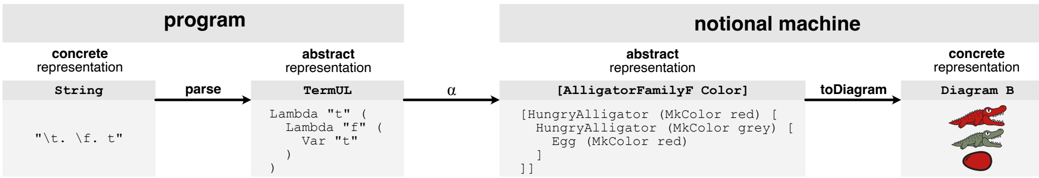


Fig. 20. Both program and notional machine have abstract and concrete representations.

We can then represent values of reference type using their locations. We could instead use arrows, like we have done in TAPLMemoryDiagram, but that would not work for `ref` 's because here we are only representing arrays.

4 ABSTRACT VS. CONCRETE SYNTAX OF NOTIONAL MACHINES

When reasoning about a notional machine, it's important to distinguish between the data types that represents the notional machine (the information present in A_{NM} and B_{NM}) and how the notional machine is visualized. This difference is akin to the distinction between the concrete syntax 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.

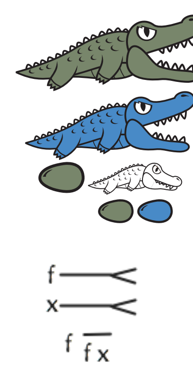


Fig. 19. Multiple concrete representations.

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 α 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). In fact, the depictions of Alligator Eggs shown here are generated by calls to our artifact that are embedded directly into the paper.

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 `λf.λx.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 not only 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 collected by Fincher et al. [2020] and classified them according to three dimensions: form, focus, and language construct. For each dimension, we present the categories of the dimension and for each category we show the number of notional machines in that category, the percentage that number represents of the total, and the sections of the paper containing notional machines that fall into that category. The notional machines are often not precesily described so we often had to make assumptions about their characteristics, how they relate to an underlying programming language, and how they are used to teach programming concepts.

The form (Table 3) of a notional machine can be metaphorical (primarely 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. We used our approach to reason about notional machines of the two forms.

Form	**Num**	**Perc.**	**Covered**
Metaphorical	22	56.41%	3.2 3.3 3.1
Diagramatic	16	41.03%	2.1 2.2 2.3 2.4
Both	1	2.56%	-
Total	37	100.00%	-

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

Most notional machines in the dataset focus (Table 4) on the runtime semantics (Evaluation) of a programming language construct (or set of conceptually related constructs) so we further break down these notional machines into the constructs they are primarely focused on (Table 5). Some entries in the table refer to sets of related constructs. For example, the category Control Flow encompasses constructs like loops and conditional statements, which primarily affect control flow. Here the classification is also not clear-cut, not only because a notional machine may focus on multiple constructs but also because some constructs are related to others. For example, notional machines that focus on functions often (although not always) represent variables as well but only the ones that solely (or primarily) focus on variables are classified as such. The category Misc includes five notional machines each of which focuses on a different construct: String literal, Procedure (side-effecting functions), Objects, Instructions (lower level operations), and one that is not clear from the description. The notional machines we reasoned about using our approach cover the majority of these dimensions and is theorectically applicable to all of them.

5.1 Expressing Complex Language Semantics

Although the notional machines presented here as well as in the dataset by Fincher et al. [2020] focus on aspects of program semantics that one may consider simple, the framework we presented can be used to reason about more complex aspects of program semantics. An example is the conceptual models to reason about Ownership Types in Rust by Crichton et al. [2023]. The authors present two models: a dynamic and a static one. 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.

Focus	Num	Perc.	Covered
Evaluation	32	82.05%	2.1 2.2 2.3 3.3 3.1
Type-checking	2	5.13%	2.4
Parsing	2	5.13%	no
Data Structure	2	5.13%	3.2
Logic Gates	1	2.56%	no
Total	37	100.00%	-

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

Construct	Num	Perc.	Covered
References	8	20.51%	2.3
Functions	5	12.82%	2.1 2.2
Variables	4	10.26%	2.1 2.2 2.3
Arrays	4	10.26%	3.3
Methods	3	7.69%	no
Control Flow	2	5.13%	no
Expressions	1	2.56%	2.1 2.2
Misc	5	12.82%	no
Total	32	82.05%	-

Table 5. Notional machines that focus on evaluation broken down by the set of construct they focus on.

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, presented 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, 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] has a nice description 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 [Roșu and Șerbănută 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

Notional machines 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 of the programming language 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.

We then evaluated how applicable our approach is to notional machines actually used in practice. Using a set of previously published notional machines used in practice, we characterize their design space and show that the notional machines we analyzed using our approach are representative of that design space.

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

DATA-AVAILABILITY STATEMENT

The artifact implementing the programming languages, notional machines, and the relationship between them as described in the paper is available at ANONYMOUS. The implementation of some of the notional machines also includes a concrete representation.

REFERENCES

Harold Abelson and Gerald Jay Sussman. 2022. *Structure and Interpretation of Computer Programs: JavaScript Edition.* MIT Press.

Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs, (Second Edition)* (second edition ed.). Vol. 33.

R. S. Bird. 1984. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems* 6, 4 (Oct. 1984), 487–504. https://doi.org/10.1145/1780.1781

R. S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (April 1989), 122–126. https://doi.org/10.1093/comjnl/32.2.122

N.G. de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, 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

Jerome Bruner. 1960. *The Process of Education.* Harvard Univesity Press.

Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. A Curated Inventory of Programming Language Misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21).* Association for Computing Machinery, New York, NY, USA, 380–386. https://doi.org/10.1145/3430665.3456343

Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (1936), 345–363. https://doi.org/10.2307/2371045 jstor:2371045

Alonzo Church. 1941. *The Calculi of Lambda-Conversion.* Number 6. Princeton University Press.

John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01).* Springer-Verlag, Berlin, Heidelberg, 320–334.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77).* Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 265:1224–265:1252. https://doi.org/10.1145/3622841

Andrew R. Dalton and William Kreahling. 2010. Automated Construction of Memory Diagrams for Program Comprehension. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10).* Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/1900008.1900040

Paul E. Dickson, Tim Richards, and Brett A. Becker. 2022. Experiences Implementing and Utilizing a Notional Machine in the Classroom. In *Proceedings of the 53rd ACM Technical Symposium V.1 on Computer Science Education (SIGCSE 2022).* Association for Computing Machinery, New York, NY, USA, 850–856. https://doi.org/10.1145/3478431.3499320

Toby Dragon and Paul E. Dickson. 2016. Memory Diagrams: A Consistant Approach Across Concepts and Languages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16).* Association for Computing Machinery, New York, NY, USA, 546–551. https://doi.org/10.1145/2839509.2844607

Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (Feb. 1986), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9 arXiv:https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Benedict Du Boulay and Tim O'Shea. 1976. *How to Work the LOGO Machine.* Technical Report 4. Department of Artificial Intelligence, University of Edinburgh.

Richard Feynman. 1985. *Surely You're Joking, Mr. Feynman!* W. W. Norton.

Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and 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

Jeremy Gibbons. 2002. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program 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

Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education - SIGCSE '13*. ACM Press, Denver, Colorado, USA, 579. https://doi.org/10.1145/2445196.2445368

C. A. Hoare. 1972. Proof of Correctness of Data Representations. *Acta Informatica* 1, 4 (Dec. 1972), 271–281. https://doi.org/10.1007/BF00289507

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

Michael Kölling and David Barnes. 2017. *Objects First With Java: A Practical Introduction Using BlueJ* (6th ed.). Pearson.

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

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

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.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, Mass.

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

Grigore Roșu and Traian Florin Șerbănută. 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

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

Juha Sorva and Teemu Sirkiä. 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

Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda*.

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

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

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