Extracting Imperative Programs from Proofs: In-place Quicksort

Ulrich Berger, Monika Seisenberger, and Gregory J. M. Woods

Swansea University
Swansea, UK
U.Berger@swansea.ac.uk, M.Seisenberger@swansea.ac.uk, csgreg@swansea.ac.uk

Abstract

The process of program extraction is primarily associated with functional programs with less focus on imperative program extraction. In this paper we consider a standard problem for imperative programming: In-place Quicksort. We formalize a proof that every array of natural numbers can be sorted and apply a realizability interpretation to extract a program from the proof. Using monads we are able to exhibit the inherent imperative nature of the extracted program. We see this as a first step towards an automated extraction of imperative programs. The case study is carried out in the interactive proof assistant Minlog.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, D.2.4 Software/Program Verification, B.5.2 Design Aids, F.4.1 Mathematical Logic

Keywords and phrases program extraction, verification, realizability, imperative programs, in-place quicksort, computational monads, Minlog

Digital Object Identifier 10.4230/LIPIcs.TYPES.2013.84

1 Introduction

Program extraction based on the proofs-as-programs paradigm is a powerful method of generating, in one step, programs together with a proof of their correctness. Often, this technique is based on some form of realizability (see e.g. [15, 5]), or a similar method, and usually yields terms denoting computable functionals or elements of a partial combinatory algebra. These terms can be naturally interpreted as functional programs. For this reason, the process of program extraction has long been associated with functional programs. There exist many tools which are able to extract functional programs from proofs (see e.g. Coq [9, 17], Minlog [18, 4], Agda [2, 8], NuPRL [22], Isabelle [13, 6]) whereas relatively little research and tool development has been explored addressing the problem of extracting imperative programs from proofs. Since most programs that are written today are more towards the imperative paradigm (cf TIOBE Index [28]) and imperative programs, in general, are notoriously difficult to verify, it would be highly desirable to have tools that allow the extraction of verified imperative programs.

In this paper we show that imperative program extraction is possible. We start with a case study where we extract an imperative In-place Quicksort algorithm from a proof. First, we informally describe In-place Quicksort, then we present a formalisation of a proof that every array can be sorted. From this proof we extract a first version of Quicksort using the Minlog system. This version of Quicksort, which is still functional, is then translated into a program that uses the well-known state monad and can be directly interpreted as the desired imperative In-place Quicksort algorithm.

An analysis of the *program* obtained from this case study leads us to a restricted functional calculus, called **SIT** (**Si**ngle-**T**hreaded Functional Language), that singles out programs

which use their array argument in a single-threaded way. All well-typed **SIT** programs can be translated into well-typed programs of a monadic language, called **MON**, that directly admits an imperative interpretation. We prove the correctness of this translation by showing that it is the inverse of the natural interpretation of **MON** in **SIT**.

The next step will be to analyse the *proof* from which our Quicksort program was extracted, and develop a proof calculus which only extracts programs in the language **SIT**, or a suitable generalisation of **SIT**, for which a similar automatic translation into imperative code is possible. We leave this, as well as the actual translation of **MON** programs into imperative code, for further work.

1.1 Related work

There are other attempts at extracting imperative programs from proofs. A notable one is carried out in [23] where the authors combine the proofs-as-programs concept with Intuitionistic Hoare Logic. However, this work leans more towards a verification strategy, which builds on the specification of the program.

A different route is Krivine's Classical Realizability ([16]) which is based on an abstract machine model that has imperative features. Krivine's work has further been adapted by Miquel [19]. The precise links between these methods and ours have yet to be investigated.

Quicksort has traditionally been a case study for many verification techniques and as such, there are many examples in the literature. A recent case study of the In-place Quicksort algorithm was formed in [11] using Event-B[1]. The behaviour of the algorithm is specified and then the algorithm is verified through a series of "refinements" where the problem is simplified into abstract machines and the effect of the abstraction makes the verification tasks more simple.

Another interesting verification is undertaken in [26] using ACL2 (see [7]) with an efficient version of the In-place Quicksort algorithm using single-threaded objects. The verification task involves showing that the efficient version of Quicksort is equivalent to a non single-threaded version and then showing that this algorithm satisfies the properties of a sorting algorithm. Once the equivalence is established, the process of verification is in some ways similar to Event-B using a "refinement" process to obtain correctness.

In relation to our work, the Event-B case study has an extraction process, through refinement, of the imperative In-place Quicksort algorithm whereas the ACL2 case study shows correctness of an already written imperative program w.r.t. the behaviour of the original functional Quicksort algorithm. The crucial difference between these works and ours is that we synthesize programs from mathematical proofs that do not require particular representations of data, and do not involve the process of programming. Therefore, our approach is highly modular, language independent, and more accessible to users outside the programming community.

1.2 Minlog

Minlog [18, 3] is an interactive proof system based on a first-order natural deduction calculus. It is not a type-theoretic system, like Coq or Agda, since it keeps formulas and proofs separate from (non-dependent) types and terms. Terms and types have a simple domain-theoretic denotational semantics. The theoretical background of Minlog is explained in [27]. One of the main motivations behind Minlog is to exploit the proofs-as-programs paradigm for program development and program verification. Minlog implements various methods of program extraction (realizability, dialectica interpretation) which also include extraction

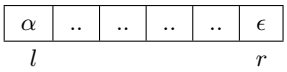
from classical proofs via the Friedman A-translation (see [25] for a comparative case study). Recent work on Minlog has been extending program extraction to simultaneous inductive and coinductive definitions and extraction to Haskell [20]. The system is supported by automatic proof search and normalization by evaluation as an efficient term rewriting device. Minlog is implemented in Scheme, and is an open system which invites users to contribute to its development and explore new methods.

2 Informal description of In-Place Quicksort

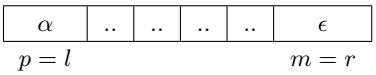
Quicksort, as we consider it in this paper, is a sorting algorithm that takes an array of natural numbers as input and sorts its elements into order of lowest to highest. The Quicksort algorithm was invented by Tony Hoare [12, p11]. Here we focus on an imperative variant of Quicksort, called In-place Quicksort, which sorts an array by repeatedly swapping elements, without using extra memory space (for creating new arrays).

In-place Quicksort works as follows:

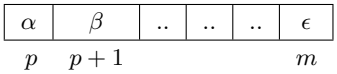
1. We are given an array a and indices l, r. We wish to sort a on the interval $[l, r] = \{i \in \mathbb{N} \mid l \leq i \leq r\}$:



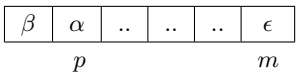
   If $l \geq r$, nothing needs to be done. Hence, in the following we assume that $l < r$.
2. Pick l as the pivot index p, and pick r as the max swap index m:



   We call $a[p]$, the element of the array at p, the *pivot*.
3. Now we compare the pivot with the element at $p + 1$:



   a. If the pivot is greater than or equal to the element at $p + 1$, then we swap the two elements of the array and increment the pivot index by 1:



   b. If the pivot is smaller than the element at $p + 1$, then we swap the element at $p + 1$ with the element at m and decrement the index m by 1:

4. Repeat step 3 until the markers p and m are equal.
5. Repeat steps 1–4 recursively on the array between l and $p - 1$ and on the array between $p + 1$ and r.

We will show that this imperative algorithm can be extracted automatically from a proof that contains no reference to imperative features at all in contrast to [23].

3 A formal proof that every array can be sorted

We sketch a formal proof that every array can be sorted. The proof is organized in such a way that its computational content corresponds to the previously described In-place Quicksort algorithm. We present the proof in some detail, so that the reader can use it as a guide through the Minlog proof script (see [24]).

For simplicity, we assume that the array to be sorted has as index set the whole set of natural numbers and every cell of the array holds a natural number (the array could hold any type of elements with a computable ordering). We view arrays as an abstract data type equipped with operations $a[i]$, for accessing the content of the array a at index i and $a[i := x]$ for changing the value of the array a at index i to x. Hence we assume the following axioms

(RW1) $b = a[i := x] \to b[i] = x\,.$
(RW2) $b = a[i := x] \wedge k \neq i \to b[k] = a[k]\,.$

Our goal is to prove that every array can be permuted into a sorted one between any given index bounds $l \leq r$

▶ **Theorem 1** (Sorting Theorem). $\forall l, r, a.\ l \leq r \to \exists b.\mathtt{Permuted}(a, b, l, r) \wedge \mathtt{Sorted}(b, l, r)$[1]

where

$$\mathtt{Sorted}(a, l, r) := \forall i, j.l \leq i < j \leq r \to a[i] \leq a[j]$$

and the intuitive meaning of the predicate `Permuted` is

$$\mathtt{Permuted}(a, b, l, r) := \exists \text{ permutation } \sigma \text{ of } [l, r].\ \forall i \in [l, r]\ b[i] = a[\sigma(i)] \quad \wedge \quad \forall i \notin [l, r].\ b[i] = a[i]$$

where $[l, r] = \{i \mid l \leq i \leq r\}$.

In the proof it will be convenient to work with a different (but equivalent) *inductive* definition of the predicate `Permuted`, based on the fact that every permutation is a composition of swappings (transpositions). First, we define what it means for two arrays a and b to differ only by swapping the contents at the indices i and j:

$$\mathtt{Swap}(a, b, i, j) := b[i] = a[j] \wedge b[j] = a[i] \wedge \forall k \notin \{i, j\}.\ b[k] = a[k]\,.$$

The predicate `Permuted` is now defined inductively by the clauses:

(P1) $\mathtt{Permuted}(a, a, l, r)\,.$
(P2) $l \leq i, j \leq r \wedge \mathtt{Swap}(a, b, i, j) \to \mathtt{Permuted}(a, b, l, r)\,.$
(P3) $\mathtt{Permuted}(a_1, a_2, l, r) \wedge \mathtt{Permuted}(a_2, a_3, l, r) \to \mathtt{Permuted}(a_1, a_3, l, r)\,.$

This means that `Permuted` is the least predicate satisfying the clauses P1–3.

We first prove we can always swap two elements of an array and that the predicate `Permuted` has the expected properties:

▶ **Lemma 2** (Swap). $\forall a, i, j.\ \exists b.\ \mathtt{Swap}(a, b, i, j)\,.$

[1] We use the dot notation where the dot allows the quantifiers to have the largest possible range. For example $\forall i.A \to B$ is the same as $\forall i(A \to B)$.

▶ **Lemma 3** (Permutation). *For all* a, b, l, r *with* $\texttt{Permuted}(a, b, l, r)$

(a) $\forall i \in [l, r].\exists j \in [l, r].\ \ b[i] = a[j]\,.$

(b) $\forall i \notin [l, r].\ \ b[i] = a[i]\,.$

(c) $\forall l', r'.\ \ l' \leq l \wedge r \leq r' \to \texttt{Permuted}(a, b, l', r')\,.$

The proofs of both lemmas are easy. The proof of the Permutation Lemma proceeds by induction on the definition of $\texttt{Permuted}(a, b, l, r)$. We omit further details.

3.1 Partitioning an array

The crucial idea of Quicksort is the notion of a *partition*. A partition is an array together with a selected pivot element such that every element to the left of the pivot is smaller than or equal to the pivot and every element to the right is greater than the pivot.

$$\begin{aligned}\texttt{Partition}(a, l, p, r) := \forall i\,(\,l \leq i < p \to a[i] \leq a[p]) \wedge \\ \forall i\,(\,p < i \leq r \to a[p] < a[i])\,.\end{aligned}$$

For the proof of the Sorting Theorem (Theorem 1) we need to prove that every array can be permuted into a partition:

▶ **Lemma 4** (Partition).

$$\forall l, r, a\ .\ l \leq r \to \exists p, b\ .\ l \leq p \leq r \wedge \texttt{Permuted}(a, b, l, r) \wedge \texttt{Partition}(b, l, p, r)\,.$$

Proof. We prove a slightly stronger statement stating in addition that $b[p] = a[l]$. The proof is by induction on $k := r - l$. Formally, we prove

$$\forall k, l, r, a\ .\ l + k = r \to \exists p \in [l, r], b\ .\ b[p] = a[l] \wedge \texttt{Permuted}(a, b, l, r) \wedge \texttt{Partition}(b, l, p, r)$$

by induction on k.

Base Case: $k = 0$. Assume $l + 0 = k$. Then $l = k$ and we may take $p = l$ and $b = a$.

Step: $k + 1$. We assume $l + (k + 1) = r$.

- Case 1: $a[l] \geq a[l + 1]\,.$
  We use the induction hypothesis with $l + 1$, r and some a_1 satisfying $\texttt{Swap}(a, a_1, l, l + 1)$, which by the Swapping Lemma exists. Since $(l + 1) + k = r$ we get $p \in [l + 1, r]$ and b with $b[p] = a_1[l + 1]$, $\texttt{Permuted}(a_1, b, l + 1, r)$ and $\texttt{Partition}(b, l + 1, p, r)$. We take the same p and b. We have $b[p] = a_1[l + 1] = a[l]$. Furthermore, by definition of the predicate Permuted, it follows that $\texttt{Permuted}(a, b, l, r)$. In order to show $\texttt{Partition}(b, l, p, r)$, we assume first $l \leq i < p$ and show $b[i] \leq b[p]$. If $i = l$, then $b[i] = a_1[l]$ by the Permutation Lemma, part (c), and hence $b[i] = a[l + 1] \leq a[l] = a_1[l + 1] = b[p]$. If $i \geq l + 1$, the in-equation $b[i] \leq b[p]$ follows from $\texttt{Partition}(b, l + 1, p, r)$. That for $p < i \leq r$ we have $b[p] < b[r]$ follows immediately from $\texttt{Partition}(b, l + 1, p, r)$.
- Case 2: $a[l] < a[l + 1]\,.$
  We use the induction hypothesis with l, $r - 1$ and a_1 satisfying $\texttt{Swap}(a, a_1, l + 1, r)$. Since $l + k = r - 1$ we get $p \in [l, r - 1]$ and b with $b[p] = a_1[l]$, $\texttt{Permuted}(a_1, b, l, r - 1)$ and $\texttt{Partition}(b, l, p, r - 1)$. Again, we take the same p and b. We have $b[p] = a_1[l] = a[l]$ since $l \notin \{l + 1, r\}$. Furthermore, by definition of the predicate Permuted, it follows $\texttt{Permuted}(a, b, l, r)$. In order to show $\texttt{Partition}(b, l, p, r)$, we assume first $p < i \leq r - 1$

and show $b[i] > b[p]$. If $i = r$, then $b[i] = a_1[r]$ by the Permutation Lemma, part (c), and hence $b[i] = a[l+1] > a[l] = b[p]$. If $i \leq r-1$, the in-equation $b[i] > b[p]$ follows from $\texttt{Partition}(b, l+1, p, r)$. That for $l \leq i < p$ we have $b[i] \leq b[p]$ follows immediately from $\texttt{Partition}(b, l, p, r-1)$.

This completes the proof of the Partition Lemma. ◀

3.2 Proof of the Sorting Theorem

Now we have everything we need to prove the Sorting Theorem (1). If $l \geq r$, the proof is trivial. Hence, in the following we assume $l < r$.

Proof. We do an induction on the length $r - l$ of the segment to be sorted. Therefore, we have an induction hypothesis for all shorter segments. Our goal is

$$\exists b.\ \texttt{Sorted}(b, l, r) \wedge \texttt{Permuted}(a, b, l, r)\,.$$

By the Partition Lemma (4) we have an array a_1 and a pivot index p that fulfil the partition property

$$\texttt{Permuted}(a, a_1, l, r) \wedge \texttt{Partition}(a_1, l, p, r)\,. \tag{1}$$

- Case 1: Both sides of the partition are non-empty, i.e. $l < p < r$.
  We use the induction hypothesis with the left side of the partition with the array $l, p-1$ and a_1 and get a_2 such that

  $$\texttt{Sorted}(a_2, l, p-1) \wedge \texttt{Permuted}(a_1, a_2, l, p-1)\,. \tag{2}$$

  We now do the same on the right side of the partition using the induction hypothesis with $p+1, r$ and a_2 and get b with:

  $$\texttt{Sorted}(b, p+1, r) \wedge \texttt{Permuted}(a_2, b, p+1, r)\,. \tag{3}$$

  We show that our b satisfies:

  $$\texttt{Sorted}(b, l, r) \wedge \texttt{Permuted}(a, b, l, r)\,.$$

  - $\texttt{Sorted}(b, l, r)$: by (2) we have $\texttt{Sorted}(a_2, l, p-1)$ and by (3), $\texttt{Sorted}(b, p+1, r)$. Furthermore, by the Permutation Lemma (3), part (b), and by using the other conjunct of (2) it follows $\texttt{Sorted}(b, l, p-1)$. Furthermore, from (1) and the Permutation Lemma (3), part (a), it follows $\texttt{Partition}(b, l, p, r)$. From these facts one easily derives $\texttt{Sorted}(b, l, r)$.
  - $\texttt{Permuted}(a, b, l, r)$: this follows easily from the Permutation Lemma, part (c), and the last clause of the inductive definition of the predicate $\texttt{Permuted}$.
- Case 2: One side of the partition is empty.
  If the left side is empty, i.e. $p = l$, then we apply the induction hypothesis to $a_1, l+1, r$. If the right side is empty we use the induction hypothesis for $a_1, l, r-1$. The rest of the proof is similar to the previous Case 1, but simpler. ◀

4 Program Extraction

In this section we address some technical issues of the implementation in Minlog, present the extracted programs and explain why they can be considered as imperative programs. The Minlog source files for this extraction example can be found on the Swansea Minlog Repository web page [24].

4.1 Implementation in Minlog

We highlight some aspects of the implementation in Minlog that are necessary to understand the proof scripts and the extracted programs.

We present the theorems as they appear in Minlog:

Listing 1 Sorting Theorem (1)

```
(set-goal (pf "all l,r,a . l<= r ->
                           ex b. Permuted a b l r & Sorted b l r"))
```

Listing 2 Partition Lemma (4)

```
(set-goal (pf "all k,l,r,a. l+k=r ->
            (ex a1,p. ((all i (l<=i ->  i<p -> Rd a1 i <= Rd a1 p)) &
                       (all i (p<i -> i<=r  ->  Rd a1 p <  Rd a1 i)) &
                       Permuted a a1 l r &
                       Rd a1 p = Rd a l &
                       l <= p &
                       p <= r))"))
```

Listing 3 Swap Lemma (2)

```
(set-goal (pf "all a,i,j ex b. Swap a b i j"))
```

4.1.1 Arrays

Minlog does not have a built-in data type of arrays. We chose to define arrays as a free algebra with the nullary constructor `Empty`, denoting the constant zero array, and a ternary constructor `Wr` (for "write"):

Listing 4 Minlog Array Definition

```
(add-alg "ar" '("Empty" "ar") '("Wr" "ar=>nat=>nat=>ar"))
```

The earlier used notation $a[i := x]$ was just syntactic sugar for $\texttt{Wr}\, a\, x\, i$.

By structural recursion on the free algebra of arrays we define a reading operation:

Listing 5 Minlog Read Definition

```
    (apc "Rd" (py "ar=>nat=>nat"))
    (add-computation-rule
      (pt "Rd (Wr a n i) j")
      (pt "[if (i=j) n (Rd a j)]"))
```

This means that `Rd` is introduced as a program constant with the computation rule rewriting `Rd (Wr a n i) j` to `[if (i=j) n (Rd a j)]`. We do not use a rewrite rule for `Rd Empty j` in the proof.

Using the notation $a[i]$ for $\texttt{Rd}\, a\, i$ the Axioms (RW1) and (RW2) (cf. Sect. 3) now become easily provable theorems.

4.1.2 Equational reasoning via normalisation

The reader might have noticed that we slightly deviated from Sect. 3 where we said that we consider arrays as an abstract data type. The reason for our choice is purely a matter of convenience since equational reasoning becomes much easier if certain equations are expressed via (strongly normalizing and confluent) term rewriting rules, allowing us to prove equations

by checking syntactic equality of normal forms. One could, of course, replace arrays by a different data structure where accessing an element takes logarithmic instead of linear time. However, this would have no effect on the extracted program.

4.1.3 Realizability

In this paper we work with Minlog's program extraction module based on modified realizability [15] which can be viewed as a typed version of Kleene's realizability for numbers [14].

Realizability provides a very intuitive way of extracting the computational content from a formal constructive proof in the spirit of the Curry-Howard correspondence that associates propositions with types and proofs with programs. It does not establish an isomorphism between proofs and programs, but a highly non-injective homomorphism that eliminates large parts of proofs that are computationally irrelevant. In order to widen the range of applications, Minlog uses an extended form of realizability that allows, for example, to extract algebraic data types from inductive definitions [18].

4.1.4 Induction and recursion

There occur two kinds of induction in our formalization:

1. Ordinary "Zero-successor"-induction on the natural numbers. This is used, for example in the proof of the Partition Lemma. In Minlog this axiom scheme is interpreted computationally (via realizability) as a constant-scheme: `Rec`.
2. The other form of induction we use is "induction with respect to a measure function". This is used in the Sorting Theorem. The realizability interpretation of this form of induction is a constant-scheme: `GRecGuard`.

4.2 The extracted programs

4.2.1 Program extracted from the proof of the Sorting Theorem.

Listing 6 Qsort Extracted Program

```
qsort =
  cGIND
  [n3,n4,g5,a6]
    [let ap7 (cPart(n4--n3)n3 n4 a6)
       [if (n3<right ap7)
           [if (right ap7<n4)
               [let a8 (g5 n3(Pred right ap7)left ap7)
                [let a9 (g5(Succ right ap7)n4 a8)
                 a9]]
               (g5 n3(Pred right ap7)left ap7)]
           [if (right ap7<n4)
               (g5(Succ right ap7)n4 left ap7)
               (left ap7)]]]
```

Note: `ap7` corresponds to (a_1, p) in the proof of the Partition Lemma. `left ap7` and `right ap7` correspond to p and a_1 respectively. `g5` plays the role of the recursive call, `cGIND` is a particular instance of the constant-scheme `GRecGuard` and `cPart` corresponds to the Partition lemma. The letter `c` in `cPart` indicates that it is an automatically generated name for the program from partition lemma `Part`. For better readability, we omit the `c` in the further discussion.

4.2.2 Program extracted from the proof of the Partition Lemma.

Listing 7 Part Extracted Program

```
part =
 [n0]
  (Rec nat=>nat=>nat=>ar=>ar@@nat)
   n0
  ([n4,n5,a6]a6@n4)
  ([n4,rec5,n6,n7,a8]
     [if (Rd a8(Succ n6)<=Rd a8 n6)
         [let a9 (Swap a8(Succ n6)n6) (rec5(Succ n6)n7 a9)]
         [let a9 (Swap a8(Succ n6)n7) (rec5 n6(Pred n7) a9)]])
```

Note: The variable `rec5` has type `nat=>nat=>ar=>ar@@nat` and represents the recursive call where `ar@@nat` is the pairing of an array and a natural number.

4.2.3 The swap function extracted from the proof of the Swapping Lemma

Listing 8 Swap Extracted Program

```
swap =
 [a0,n1,n2]
   [let n3 (Rd a0 n1)
     [let n4 (Rd a0 n2)
       [let a1 (Wr a0 n4 n1) (Wr a1 n3 n2)]]]
```

4.3 The extracted programs explained

Minlog's formalization of recursive definitions via recursion operators is adequate from a technical point of view, but it makes recursively defined functions hard to read. Therefore, we replace the recursion operators by recursive equations, and re-name the (automatically generated) variables so that they match the variable names used in the proofs. Furthermore, we write the variable `ap7` in the let expression of `qsort` as a pair `(a1,p)` which spares us the use of the projection functions `left` and `right`. Also, in the function `part` we omit the first parameter `n0`, which corresponds to k, since it always has the value $r - l$ (`n4-n5`), and replace, for example "$k = 0$" by $l = r$. All these changes are only cosmetic and intended to ease the understanding of the programs. They do not affect their behaviour.

We use Haskell-like syntax in the pseudo-code below. For example, `(N,N,A)` stands for the type $\mathbb{N} \times \mathbb{N} \times A$ where A is the type of arrays. The programs `qsort` and `part` are intended to be used for $l \leq r$ only.

Listing 9 Qsort (Haskell Style)

```
qsort(l,r,a) =
  let (a1,p) = part(l,r,a)
  in if l < p
     then if p < r
          then let {a2 = qsort(l,p-1,a1)} in qsort(p+1,r,a2)
          else qsort(l,r-1,a1)
     else if p < r
          then qsort(l+1,r,a1)
          else a1
```

Listing 10 Part (Haskell Style)

```
part : (N,N,A) -> (A,N)
part(l,r,a) =
  if l = r
  then (a,l)
  else if a[l+1] <= a[l]
       then let {a1 = swap(a,l+1,l)} in part(l+1,r,a1)
       else let {a1 = swap(a,l+1,r)} in part(l,r-1,a1)
```

Listing 11 Swap (Haskell Style)

```
swap : (A,N,N) -> A
swap(a,i,j) = let {x = a[i]; y = a[j]; a1 = a[i:=y]} in a1[j:=x]
```

4.4 How are these extracted programs imperative?

At first sight the extracted programs look clearly functional. Indeed they are, provided the writing operation `a[i:=y]`, which is the only operation where the array is modified, is implemented functionally. However, nothing prevents us from implementing the write operation as a procedure that destructively changes the array (instead of producing a new array). But, are the extracted programs then still correct? Looking at the proof of the Soundness Theorem (see eg [5]) for realizability, which is the source of correctness of our extracted program, we see that the extracted programs are assumed to behave functionally, i.e. have no side effects. In particular, functional programs do not destroy their input, and as a consequence, referential transparency holds, which means that the value of a complex programs only depends on the values of its subprograms and not, for example, on the order in which the subprograms are evaluated. Referential transparency is crucial for the Soundness Theorem, and it does, in general, not hold for imperative code. Therefore, an additional argument is needed in order to show that, in our particular case, the programs stay correct when the writing operation is implemented imperatively. The argument is simple: in the extracted programs, arrays can be viewed as "single-threaded" objects, since, once an array is used as an argument of the update operation, or a program that uses the writing operation such as `swap`, `part` or `qsort`, it is never used again. Hence the correctness of the program is not compromised if the write operation destroys its argument.

4.5 Monadic presentation of the extracted programs

The imperative nature of the programs `qsort` and `part` becomes particularly lucid when they are formulated using the *state monad* where arrays play the role of states. Monads [21] are a popular concept to incorporate imperative code into functional programs in an elegant and clean way. The definitions below are standard in functional programming, but in order to make the paper self-contained we include them.

We define a type operator `M` (the state monad with arrays as states) by

```
M u = A -> (u,A)
```

where `u` is a type variable. A value of type `M u` can be viewed as an *action* that, when executed, produces a result of type `u`, but may, in addition, have a side effect on the state (array in our case).

We have the general monad operators

```
return : u -> M u
return x a = (x,a)
```

```
bind : M u -> (u -> M v) -> M v
bind m f a = let {(x,b) = m a} in f x b
```

and the special operators for this particular monad:

```
get : N -> M N
get i a = (a[i],a)
```

```
put : (N,N) -> M ()
put(i,x) a = ((),a[i:=x])
```

where `()` is a singleton type, We will use the suggestive "do-notation": If `m1 : M u` and `m2 : M v` are expressions where `m2` may depend on a variable `x : u`, then

```
do { x <- m1 ; m2 } := bind m1 (\x-> m2) : M v
```

where `\x->` denotes lambda-abstraction. If `m2` does not depend on `x`, then

```
do { m1 ; m2 } := bind m1 (\_ -> m2) : M v
```

This notation can be extended to more than two actions in the obvious way. Furthermore, expressions of the form `do { ... x <- m ; return x }` can be simplified to `do { ... m }`.

Using canonical isomorphisms such as

$$\mathbb{N} \times \mathbb{N} \times A \to A \quad \simeq \quad \mathbb{N} \times \mathbb{N} \to (A \to () \times A)$$

we can write the programs `qsort`, `part`, and `swap` equivalently as follows, using the new names `mqsort`, `mpart` and `mswap`:

Listing 12 Qsort (Monadic Style)

```
mqsort(l,r) =
  do {
       p <- mpart(l,r) ;
       (if l < p
        then if p < r
             then do { mqsort(l,p-1) ; mqsort(p+1,r) }
             else mqsort(l,r-1)
        else if p < r
             then mqsort(l+1,r)
             else return ()
     }
```

Listing 13 Part (Monadic Style)

```
mpart : (N,N) -> M N
mpart(l,r) =
  if l = r
  then return l
  else do {
            x <- get l ;
            y <- get (l+1) ;
            (if x >= y
             then do { mswap(l,l+1); mpart(l+1,r) }
             then do { mswap(l+1,r); mpart(l,r-1) }
          }
```

Listing 14 Swap (Monadic Style)

```
mswap : (N,N) -> M ()
mswap(i,j) = do { x <- get(i) ; y <- get(j) ; put(i,y) ; put(j,x) }
```

The significance of the monadic notation is that the syntax alone enforces that arrays, once they have been used as input to a program involving the write operation, can no longer be accessed, thus enabling destructive interpretations of these operations. For example, we might want a modified sorting program `qsort1` that outputs the sorted array together with the original array. In the non-monadic style this can be done by simply defining

```
qsort1(l,r,a) = (qsort(l,r,a),a)
```

This would exhibit the desired behaviour if `qsort` is interpreted functionally, but not if it is interpreted imperatively. However, in the monadic style we need a copying operation

```
copy : M A
copy a = (a,a)
```

in order to define

```
mqsort1(l,r) = do { a <- copy ; mqsort(l,r) ; return a }
```

The latter program is correct w.r.t. the functional *and* the imperative interpretation.

5 Automated Monadification

Abstracting from the Quicksort case study we now define a functional language **SIT** modelling a subset of Minlog's term language which captures the idea of single-threadedness and includes the original functional Quicksort program extracted by Minlog. Then we describe a (meaning preserving) translation of this language into a monadic language which admits an imperative interpretation.

5.1 Remark

A different monadification process has been studied by Erwig and Ren [10]. They consider the problem of transforming a program of type $\rho \to \sigma$ into one of type $\rho \to M(\sigma)$ where M is a "runnable" monad (i.e. a monad with a left inverse of the *return* operation) and the resulting program is again functional. On the other hand, our translation transforms programs of type $(\rho, A) \to (\sigma, A)$ into programs of type $\rho \to M(\sigma)$ where M is specifically the state monad, i.e. we deal with programs where the input and output may be state dependent. In addition we are careful that the resulting program has an imperative interpretation.

5.2 The Single-Threaded Functional Language SIT

In the **SIT** language we have two different kinds of types. A distinguished type A of *states* and other "ordinary" types ρ, σ different from A. For the Quicksort example the state is an array and the only other types are natural numbers and Boolean values.

SIT is parametrized by three different kinds of functions which are distinguished by the types of values they access and return:

- Basic Functions – functions that neither access or modify the state:
  - $f^{\mathrm{bas}} : \vec{\rho} \to \sigma$
- Accessor Functions – functions which may require access to the state, but do *not* modify it:
  - $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$

- Quasi Side-Effect Functions – functions that may modify the state ("Quasi" because they are **SIT** functions which are side-effect free, but will later be translated into functions with side-effects):
  - $f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A)$

In $(\rho_1, \ldots, \rho_n, A)$ the number n may be zero in which case we write just (A). We let x, y, z range over an infinite set of variables which will later be given ordinary types. We only need *one* state variable a.

The terms t and expressions e of **SIT** are defined as follows:

- SitTerm $\ni t ::= x \mid f^{\mathrm{bas}}(\vec{t}) \mid f^{\mathrm{acc}}(\vec{t}, a)$
- SitExpr $\ni e ::= (\vec{t}, a) \mid f^{\mathrm{sid}}(\vec{t}, a) \mid$ if t then e_1 else $e_2 \mid$ let $(\vec{x}, a) = e_1$ in e_2

where all variables in $\vec{x}$ are different. In general, $\vec{x}, \vec{y}, \ldots$ will always denote vectors of pairwise different variables.

A **SIT** program is a finite list of equations of the form

$$f_1^{\mathrm{sid}}(\vec{x}_1, a) = e_1$$
$$\ldots$$
$$f_n^{\mathrm{sid}}(\vec{x}_n, a) = e_n\,.$$

The functions f_i^{sid} may occur in the expressions e_j and are considered to be (possibly recursively) defined by the equations. All other functions symbols occurring in the e_j are considered as predefined functions (or parameters of the program).

The "let" is the only construction which binds free variables. In fact, the "let" expression let $(\vec{x}, a) = e_1$ in e_2 is to be understood as the β-redex $(\lambda(\vec{x}, a).e_2)e_1$. This intuition is reflected by the following axiom:

▶ Axiom 1. let $(\vec{x}, a) = (\vec{t}, a)$ in $e \equiv e[\vec{t}/\vec{x}]$ for all $\vec{t} \in$ SitTerm, $e \in$ SitExpr,

where substitution of terms into expressions, $e[\vec{t}/\vec{x}]$, is defined below.

Note that in a "let" expression, let $(\vec{x}, a) = e_1$ in e_2, the equation $(\vec{x}, a) = e_1$ is *not* to be understood as a (potentially) recursive definition of $(\vec{x}, a)$ as it is the case in Haskell. Therefore, our "let" rather corresponds to Scheme's "let" while Haskell's "let" corresponds to Scheme's "letrec".

The free variables for "let" constructs are defined as:

$$\mathrm{FV}(\text{let } (\vec{x}, a) = e_1 \text{ in } e_2) := \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \{\vec{x}, a\})\,.$$

The notion of α-equivalence is defined as usual, in particular:

$$\text{let } (\vec{x}, a) = e_1 \text{ in } e_2 \overset{\alpha}{\equiv} \text{let } (\vec{y}, a) = e_1 \text{ in } e_2[\vec{y}/\vec{x}]$$

where $\vec{y}$ are fresh variables. Simultaneous substitution, $e[\vec{t}/\vec{x}]$, of terms $\vec{t}$ for variables $\vec{x}$ in a **SIT** expression e is defined as follows:

$$y[\vec{t}/\vec{x}] := \begin{cases} t_i, & \text{if } y \in \{x_1, \ldots, x_n\}, \text{where } y = x_i \\ y, & \text{otherwise} \end{cases}$$
$$f^{\mathrm{bas}}(\vec{s})[\vec{t}/\vec{x}] := f^{\mathrm{bas}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}])$$
$$f^{\mathrm{acc}}(\vec{s}, a)[\vec{t}/\vec{x}] := f^{\mathrm{acc}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a)$$
$$(\vec{s}, a)[\vec{t}/\vec{x}] := (s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a)$$
$$f^{\mathrm{sid}}(\vec{s}, a)[\vec{t}/\vec{x}] := f^{\mathrm{sid}}(s_1[\vec{t}/\vec{x}], \ldots, s_n[\vec{t}/\vec{x}], a)$$
$$(\text{if } t \text{ then } e_1 \text{ else } e_2)[\vec{t}/\vec{x}] := \text{if } s[\vec{t}/\vec{x}] \text{ then } e_1[\vec{t}/\vec{x}] \text{ else } e_2[\vec{t}/\vec{x}]$$
$$(\text{let } (\vec{y}, a) = e_1 \text{ in } e_2)[\vec{t}/\vec{x}] := \text{let } (\vec{y}, a) = e_1[\vec{t}/\vec{x}] \text{ in } e_2[\vec{t}/\vec{x}]$$

where in the "let" case we can assume, possibly after α-renaming, that $\vec{y} \cap (\vec{x} \cup \mathrm{FV}(\vec{t})) = \emptyset$.

5.3 SIT Typing System

In the following we define a typing system which derives judgements of the form $\Gamma \vdash t : \rho$ or $\Gamma \vdash e : (\vec{\rho}, A)$ where the context Γ is a finite set of type declarations $x : \rho$:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{f^{\mathrm{bas}} : \vec{\rho} \to \sigma \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{bas}}(\vec{t}) : \sigma}$$

$$\frac{f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{acc}}(\vec{t}, a) : \sigma}$$

$$\frac{\Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash (\vec{t}, a) : (\vec{\rho}, A)}$$

$$\frac{f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A) \qquad \Gamma \vdash \vec{t} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{sid}}(\vec{t}, a) : (\vec{\sigma}, A)}$$

$$\frac{\Gamma \vdash t : \mathbb{B} \qquad \Gamma \vdash e_1 : (\vec{\sigma}, A) \qquad \Gamma \vdash e_2 : (\vec{\sigma}, A)}{\Gamma \vdash \text{if } t \text{ then } e_1 \text{ else } e_2 : (\vec{\sigma}, A)}$$

$$\frac{\Gamma \vdash e_1 : (\vec{\rho}, A) \qquad \Gamma, \vec{x} : \vec{\rho} \vdash e_2 : (\vec{\sigma}, A)}{\Gamma \vdash \text{let } (\vec{x}, a) = e_1 \text{ in } e_2 : (\vec{\sigma}, A)}$$

A **SIT** program

$$\begin{aligned} f_1^{\mathrm{sid}}(\vec{x}_1, a) &= e_1 \\ &\cdots \\ f_n^{\mathrm{sid}}(\vec{x}_n, a) &= e_n \end{aligned}$$

is well-typed if for each of the defined functions $f_i^{\mathrm{sid}} : (\vec{\rho}_i, A) \to (\vec{\sigma}_i, A)$ the typing judgement $\vec{x_i} : \vec{\rho}_i \vdash e_i : (\vec{\sigma}_i, A)$ is derivable.

Since the typing rules are syntax directed it is clear that the typing rules are decidable (in linear time) and therefore can be automated.

The single-threadedness of **SIT** is enforced by the fact that there are no functions with result type A. For example, `Wr` has result type (A) but not A. Otherwise, one could form well-typed, but non single-threaded terms such as $\mathtt{Rd}(i, \mathtt{Wr}(i, x, a)) + \mathtt{Rd}(i, a)$.

5.4 Quicksort as a SIT program

We briefly demonstrate that the extracted Quicksort program (written as recursive equations) can be written in **SIT**. We have the predefined functions $<, \leq, = : (\mathbb{N}, \mathbb{N}) \to \mathbb{B}$ (basic functions), $\mathtt{Rd} : (\mathbb{N}, A) \to \mathbb{N}$ (accessor function) and $\mathtt{Wr} : (\mathbb{N}, \mathbb{N}, A) \to (A)$ (quasi side-effect function). Semantically (expressed outside the **SIT** language), $\mathtt{Rd}(i, a) = a[i]$ and $\mathtt{Wr}(i, x, a) = (a[i := x])$. The **SIT** program defining Quicksort is as follows:

Listing 15 Qsort (**SIT**)

```
qsort : (N,N,A) -> (A)
qsort(l,r,a) =
  let (p,a) = part(l,r,a) in
     if l < p
     then if p < r
          then let (a) = qsort(l,p-1,a) in qsort(p+1,r,a)
          else qsort(l,r-1,a)
     else if p < r
          then qsort(l+1,r,a)
          else (a)
```

Listing 16 Part (**SIT**)

```
part : (N,N,A) -> (N,A)
part(l,r,a) =
  if l = r
  then (l,a)
  else if Rd(l+1,a) <= Rd(l,a)
       then let (a) = swap(l+1,l,a) in part(l+1,r,a)
       else let (a) = swap(l+1,r,a) in part(l,r-1,a)
```

Listing 17 Swap (**SIT**)

```
swap : (N,N,A) -> (A)
swap(i,j,a) = let (x,y,a) = (Rd(i,a),Rd(j,a),a) in
                let (a) = Wr(i,y,a) in Wr(j,x,a)
```

The above **SIT** code wouldn't run correctly in Haskell because, as mentioned in Sect. 5.2, Haskell would interpret, for example, `let (p,a) = part(l,r,a)` as a recursive definition of `(p,a)`. In order to obtain correct Haskell code (up to minor syntactic details such as capitalised functions) we can simply α-rename the state variables, for example, rename `let (p,a) = part(l,r,a) in` e to `let (p,a1) = part(l,r,a) in` $e[a1/a]$. Then we would end up with essentially the same code as in listings 9–11.

5.5 The Monadic Language MON

We now introduce a monadic language **MON** with programs of type $\vec{\rho} \to M\vec{\sigma}$ where M is a monad. For convenience we let M operate on tuples of types rather than single types. For the moment M is an arbitrary monad, but later in Sect. 6.1 we will interpret **MON** into **SIT** where it will be pinned down as the state monad. In **MON** we have two different kinds of functions:

- The basic functions of the **SIT** language,
$$f^{\mathrm{bas}} : \vec{\rho} \to \sigma\,.$$
- Functions which may access and modify the state,
$$g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma}\,.$$

The terms u and expressions m of **MON** are defined as follows:

- MonTerm $\ni u ::= x \mid f^{\mathrm{bas}}(\vec{u})$
- MonExpr $\ni m ::= g^{\mathrm{sid}}(\vec{u}) \mid \mathrm{return}\ \vec{u} \mid \mathrm{if}\ u\ \mathrm{then}\ m_1\ \mathrm{else}\ m_2 \mid \mathrm{bind}\ m_1\ (\lambda\vec{x}.m_2)$

where in bind m_1 $(\lambda\vec{x}.m_2)$ we assume $\vec{x} \notin \mathrm{FV}(m_1)$.

Notice that MonTerm $\subset$ SitTerm. A **MON** program is a finite list of equations

$$\begin{aligned} g_1^{\mathrm{sid}}(\vec{x}_1) &= m_1 \\ &\dots \\ g_n^{\mathrm{sid}}(\vec{x}_n) &= m_n\,. \end{aligned}$$

As with **SIT**, the functions g_i^{sid} are considered to be (possibly recursively) defined by the equations. All other functions symbols occurring in the m_j are considered as predefined functions.

5.6 MON Typing System

We define a typing system for judgements of the form $\Gamma \vdash u : \sigma$ and $\Gamma \vdash m : M\vec{\sigma}$ where contexts Γ are as before:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{f^{\mathrm{bas}} : \vec{\rho} \to \sigma \qquad \Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash f^{\mathrm{bas}}(\vec{u}) : \sigma}$$

$$\frac{\Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash \mathrm{return}\ \vec{u} : M\vec{\rho}}$$

$$\frac{g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma} \qquad \Gamma \vdash \vec{u} : \vec{\rho}}{\Gamma \vdash g^{\mathrm{sid}}(\vec{u}) : M\vec{\sigma}}$$

$$\frac{\Gamma \vdash u : \mathbb{B} \qquad \Gamma \vdash m_1 : M\vec{\sigma} \qquad \Gamma \vdash m_2 : M\vec{\sigma}}{\Gamma \vdash \mathrm{if}\ u\ \mathrm{then}\ m_1\ \mathrm{else}\ m_2 : M\vec{\sigma}}$$

$$\frac{\Gamma \vdash m_1 : M\vec{\rho} \qquad \Gamma, \vec{x} : \vec{\rho} \vdash m_2 : M\vec{\sigma}}{\Gamma \vdash \mathrm{bind}\ m_1\ (\lambda\vec{x}.m_2) : M\vec{\sigma}}$$

5.7 Translation from SIT to MON

The translation of **SIT** into **MON** is parametric in a translation of the predefined function symbols. Hence, we assume that we have assigned in a one-to-one way:

- to every **SIT** accessor function $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$ a **MON** side-effect function $\mathbf{F}_0(f^{\mathrm{acc}}) : \vec{\rho} \to M\sigma$, and
- to every **SIT** quasi side-effect function $f^{\mathrm{sid}} : (\vec{\rho}, A) \to (\vec{\sigma}, A)$ a **MON** side-effect function $\mathbf{F}_0(f^{\mathrm{sid}}) : \vec{\rho} \to M\vec{\sigma}$

Recall that basic **SIT** functions $f^{\mathrm{bas}} : \vec{\rho} \to \sigma$ are at the same time basic **MON** functions and are hence translated into themselves. Based on this assignment we define a translation

$$\mathbf{F} : \mathrm{SitTerm} \cup \mathrm{SitExpr} \to \mathrm{MonExpr}$$

Terms

$\mathbf{F}(x)$	:=	return x (where x is a variable)
$\mathbf{F}(f^{\mathrm{bas}}(\vec{t}))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}$.return $f^{\mathrm{bas}}(\vec{x})$
$\mathbf{F}(f^{\mathrm{acc}}(\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.\mathbf{F}_0(f^{\mathrm{acc}})(\vec{x})$

Where bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.m$ is defined recursively as:

bind $\mathbf{F}(\emptyset)$ $\lambda\emptyset.m := m$

bind $\mathbf{F}(t, \vec{t})$ $\lambda x.\lambda\vec{x}.m :=$ bind $\mathbf{F}(t)$ $(\lambda x.$bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.m)$

assuming $\vec{x} \cap \mathrm{FV}(\vec{t}) = \emptyset$

Expressions

$\mathbf{F}((\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}$.return $\vec{x}$
$\mathbf{F}(f^{\mathrm{sid}}(\vec{t}, a))$	:=	bind $\mathbf{F}(\vec{t})$ $\lambda\vec{x}.\mathbf{F}_0(f^{\mathrm{sid}})(\vec{x})$
$\mathbf{F}(\mathrm{if}\ t\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2)$	:=	bind $\mathbf{F}(t)$ $(\lambda x.$if x then $\mathbf{F}(e_1)$ else $\mathbf{F}(e_2))$
$\mathbf{F}(\mathrm{let}(\vec{x}, a) = e_1\ \mathrm{in}\ e_2)$	:=	bind $\mathbf{F}(e_1)$ $(\lambda\vec{x}.\mathbf{F}(e_2))$

where all introduced λ-abstractions use fresh variables. This induces a translation of any **SIT** program:

$$f_1^{\mathrm{sid}}(\vec{x}_1, a) = e_1$$
$$\dots$$
$$f_n^{\mathrm{sid}}(\vec{x}_n, a) = e_n$$

into the **MON** program:

$$\mathbf{F}_0(f_1^{\mathrm{sid}})(\vec{x}_1) = \mathbf{F}(e_1)$$
$$\dots$$
$$\mathbf{F}_0(f_n^{\mathrm{sid}})(\vec{x}_n) = \mathbf{F}(e_n)\,.$$

▶ **Lemma 5** (Type preservation of translation **F**).
- *If* $\Gamma \vdash t : \rho$ *then* $\Gamma \vdash \mathbf{F}(t) : M\rho$, *and*
- *If* $\Gamma \vdash e : (\vec{\rho}, A)$ *then* $\Gamma \vdash \mathbf{F}(e) : M\vec{\rho}$

Proof. By induction on the typing derivations. ◀

▶ **Corollary 6.** *If a* **SIT** *program is well-typed then its monadic translation is also well-typed.*

5.8 Translating the SIT code of Quicksort into MON

We apply the translation **F** to the **SIT** code of the Quicksort program shown in Sect. 5.4. Defining the function $\mathbf{F}_0$ as:
- $\mathbf{F}_0(\texttt{Rd}) = \texttt{get}$
- $\mathbf{F}_0(\texttt{Wr}) = \texttt{put}$
- $\mathbf{F}_0(\texttt{qsort}) = \texttt{mqsort}$
- $\mathbf{F}_0(\texttt{part}) = \texttt{mpart}$
- $\mathbf{F}_0(\texttt{swap}) = \texttt{mswap}$

and using the "do-notation" introduced in Sect. 4.5 we obtain exactly the **MON** programs in the same section (listings 12-14).

6 Soundness of the translation

To prove the soundness of the monadic translation **F** we define a new translation:

$$\mathbf{G} : \mathbf{MON} \to \mathbf{SIT}$$

which "demonadifys" the program. We will prove the following theorem (which will be Theorem 10 below):

For all typable **SIT** expressions e, $\mathbf{G}(\mathbf{F}(e)) \equiv e$.

6.1 Translation from MON to SIT

The following translation of **MON** into **SIT** can be viewed as a definition of the monadic constructs in functional terms. In fact, it expresses that M is the state monad. To every **MON** side-effect function $g^{\mathrm{sid}} : \vec{\rho} \to M\vec{\sigma}$, each of which we may assume to be in the image of $\mathbf{F}_0$, we assign a **SIT** quasi side-effect function $\mathbf{G}_0(g^{\mathrm{sid}}) : (\vec{\rho}, A) \to (\vec{\sigma}, A)$ as follows:
- If $g^{\mathrm{sid}} = \mathbf{F}_0(f^{\mathrm{sid}})$, then $\mathbf{G}_0(g^{\mathrm{sid}}) := f^{\mathrm{sid}}$,
- If $g^{\mathrm{sid}} = \mathbf{F}_0(f^{\mathrm{acc}})$, then $\mathbf{G}_0(g^{\mathrm{sid}}) := \widehat{f^{\mathrm{acc}}}$,

where for a **SIT** accessor function $f^{\mathrm{acc}} : (\vec{\rho}, A) \to \sigma$, $\widehat{f^{\mathrm{acc}}} : (\vec{\rho}, A) \to (\sigma, A)$ is a new **SIT** quasi side-effect function that "behaves" like f^{acc}, that is, we postulate:

▶ Axiom 2. $\widehat{f^{\mathrm{acc}}}(\vec{x}, a) \equiv (f^{\mathrm{acc}}(\vec{x}, a), a)$.

Now we define the function

$$\mathbf{G} : \mathrm{MonExpr} \to \mathrm{SitExpr}\,.$$

Recall that MonTerm $\subset$ SitTerm, and **SIT** has only one state variable a which is used below.

$$\begin{array}{lcl}
\mathbf{G}(g^{\mathrm{sid}}(\vec{u})) & := & \mathbf{G}_0(g^{\mathrm{sid}})(\vec{u}, a) \\
\mathbf{G}(\text{return } \vec{u}) & := & (\vec{u}, a) \\
\mathbf{G}(\text{if } u \text{ then } m_1 \text{ else } m_2) & := & \text{if } u \text{ then } \mathbf{G}(m_1) \text{ else } \mathbf{G}(m_2) \\
\mathbf{G}(\text{bind } m_1\ \lambda\vec{x}.m_2) & := & \text{let } (\vec{x}, a) = \mathbf{G}(m_1) \text{ in } \mathbf{G}(m_2)
\end{array}$$

6.2 Auxiliary Lemmas

We prepare the proof of Theorem 10 by a sequence of lemmas.

▶ **Lemma 7.** *Let $\vec{t} = t_1, \ldots, t_n$. Assume $\Gamma \vdash \vec{t} : \vec{\rho}$, and $\mathbf{G}(\mathbf{F}(t_i)) \equiv (t_i, a)$. Let $\vec{x} = x_1, \ldots, x_n$ be pairwise different variables that are not free in $\vec{t}$. Then:*
(a) $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\text{return } f^{bas}(\vec{x})) \equiv (f^{bas}(\vec{t}), a)$
(b) $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\mathbf{F}_0(f^{acc})(\vec{x})) \equiv (f^{acc}(\vec{t}, a), a)$

Proof. For part (a) we prove more generally:

$$\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1}, \ldots, t_n)\ \lambda x_{k+1}, \ldots, x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \equiv (f^{\mathrm{bas}}(x_1, \ldots, x_k, t_{k+1}, \ldots, t_n), a)$$

for all $k \in 1, \ldots, n$, by induction on $n - k$.

- Base case $n - k = 0$ $(n = k)$.

$$\begin{array}{ll}
& \mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \mathbf{G}(\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \qquad \text{(def. of bind)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_n), a) \qquad \text{(def. of } \mathbf{G})
\end{array}$$

- Step $n > 0$ $(k < n)$.

$$\begin{array}{ll}
& \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1}, \ldots, t_n)\ \lambda x_{k+1}, \ldots, x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1})\ (\lambda x_{k+1}.\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n) \qquad \text{(def. of bind)} \\
& \qquad \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n))) \\
\equiv & \text{let}(x_{k+1}, a) = \mathbf{G}(\mathbf{F}(t_{k+1})) \text{ in} \qquad \text{(def. of } \mathbf{G}) \\
& \quad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n)\ \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \text{let}(x_{k+1}, a) = (t_{k+1}, a) \text{ in} \qquad \text{(assumption)} \\
& \quad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2}, \ldots, t_n)\ \lambda x_{k+2}, \ldots, \lambda x_n.\text{return } f^{\mathrm{bas}}(x_1, \ldots, x_n)) \\
\equiv & \text{let } (x_{k+1}, a) = (t_{k+1}, a) \text{ in } (f^{\mathrm{bas}}(x_1, \ldots, x_k, x_{k+1}, t_{k+2}, \ldots, t_n), a) \qquad \text{(I.H.)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_k, x_{k+1}, t_{k+2}, \ldots, t_n), a)[t_{k+1}/x_{k+1}] \qquad \text{(Axiom 1)} \\
\equiv & (f^{\mathrm{bas}}(x_1, \ldots, x_k, t_{k+1}, t_{k+2}, \ldots, t_n), a) \qquad (x_{k+1} \notin \mathrm{FV}(\vec{t}))
\end{array}$$

Similarly for part (b) we prove the following more general statement:

$$\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1},\ldots,t_n)\ \lambda x_{k+1},\ldots,x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)) \equiv (f^{\text{acc}}(x_1,\ldots,x_k,t_{k+1},\ldots,t_n,a),a)$$

for all $k \in 1,\ldots,n$, by induction on $n-k$.

- Base case $n-k=0$ $(n=k)$.

$$\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.\ \mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \mathbf{G}(\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)) && \text{(def. of bind)}\\
&\equiv \mathbf{G}_0(\mathbf{F}_0(f^{\text{acc}}))(x_1,\ldots,x_n,a) && \text{(def. of } \mathbf{G})\\
&\equiv \widehat{f^{\text{acc}}}(x_1,\ldots,x_n,a) && \text{(def. of } \mathbf{G}_0)\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_n,a),a) && \text{(Axiom 2)}
\end{aligned}$$

- Step $n>0$ $(k<n)$.

$$\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(t_{k+1},\ldots,t_n)\ \lambda x_{k+1},\ldots,x_n.\ \mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t_{k+1})\ (\lambda x_{k+1}.\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n) && \text{(def. of bind)}\\
&\qquad\qquad \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n)))\\
&\equiv \text{let}(x_{k+1},a) = \mathbf{G}(\mathbf{F}(t_{k+1})) \text{ in} && \text{(def. of } \mathbf{G})\\
&\qquad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n)\ \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \text{let}(x_{k+1},a) = (t_{k+1},a) \text{ in} && \text{(assumption)}\\
&\qquad \mathbf{G}(\text{bind } \mathbf{F}(t_{k+2},\ldots,t_n)\ \lambda x_{k+2},\ldots,\lambda x_n.\mathbf{F}_0(f^{\text{acc}})(x_1,\ldots,x_n))\\
&\equiv \text{let } (x_{k+1},a) = (t_{k+1},a) \text{ in } (f^{\text{acc}}(x_1,\ldots,x_k,x_{k+1},t_{k+2},\ldots,t_n,a),a) && \text{(I.H.)}\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_k,x_{k+1},t_{k+2},\ldots,t_n,a),a)[t_{k+1}/x_{k+1}] && \text{(Axiom 1)}\\
&\equiv (f^{\text{acc}}(x_1,\ldots,x_k,t_{k+1},t_{k+2},\ldots,t_n,a),a) && (x_{k+1} \notin \text{FV}(\vec{t}))
\end{aligned}$$

◀

▶ **Lemma 8** (Soundness for a single term)**.**
If $\Gamma \vdash t : \sigma$ *(i.e.* t *is typable in the* **SIT** *language), then* $\mathbf{G}(\mathbf{F}(t)) = (t,a)$.

Proof. By induction on **SIT** terms.

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(x))\\
&\equiv \mathbf{G}(\text{return } x) && \text{(def. of } \mathbf{F})\\
&\equiv (x,a) && \text{(def. of } \mathbf{G})
\end{aligned}$$

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\text{bas}}(\vec{s}))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{s})\ \lambda\vec{x}.\text{return } f^{\text{bas}}(\vec{x})) && \text{(def. of } \mathbf{F})\\
&\equiv (f^{\text{bas}}(\vec{s}),a) && \text{(Lemma 7)}
\end{aligned}$$

$$\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\text{acc}}(\vec{s},a)))\\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{s})\ \lambda\vec{x}.\ \mathbf{F}_0(f^{\text{acc}})(\vec{x})) && \text{(def. of } \mathbf{F})\\
&\equiv (f^{\text{acc}}(\vec{x},a),a) && \text{(Lemma 7)}
\end{aligned}$$

◀

▶ **Lemma 9** (Bind Lemma). *If* $\Gamma \vdash \vec{t} : \vec{\rho}$ *and* $\vec{x} \cap \mathrm{FV}(\vec{t}) = \emptyset$, *then* $\mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m) \equiv \mathbf{G}(m)[\vec{t}/\vec{x}]$.

Proof. By induction on the length of $\vec{t}$.

- Base $\mathrm{len}(\vec{t}) = 0$ (i.e. $\vec{t} = \emptyset$)

$$
\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(\emptyset)\ \lambda\emptyset.m) \\
&\equiv \mathbf{G}(m) && \text{(def. of bind)}
\end{aligned}
$$

- Step

$$
\begin{aligned}
&\mathbf{G}(\text{bind } \mathbf{F}(t,\vec{t})\ \lambda x\lambda\vec{x}.m) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t)\ (\lambda x.\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m)) && \text{(def. of bind)} \\
&\equiv \text{let } (x,a) = \mathbf{G}(\mathbf{F}(t)) \text{ in } \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.m) && \text{(def. of } \mathbf{G}) \\
&\equiv \text{let } (x,a) = \mathbf{G}(\mathbf{F}(t)) \text{ in } (\mathbf{G}(m)[\vec{t}/\vec{x}]) && \text{(I.H.)} \\
&\equiv \text{let } (x,a) = (t,a) \text{ in } (\mathbf{G}(m)[\vec{t}/\vec{x}]) && \text{(Lemma 8)} \\
&\equiv \mathbf{G}(m)[\vec{t}/\vec{x}][t/x] && \text{(Axiom 1)} \\
&\equiv \mathbf{G}(m)[t,\vec{t}/x,\vec{x}] && (x \notin \mathrm{FV}(\vec{t}))
\end{aligned}
$$

◀

6.3 Soundness Proof

Now we show the main Theorem:

▶ **Theorem 10** (Soundness for expressions). *If* $\Gamma \vdash e : (\vec{\rho}, A)$, *then* $\mathbf{G}(\mathbf{F}(e)) \equiv e$

Proof. By induction on **SIT** expressions.

- $e \equiv (\vec{t}, a)$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}((\vec{t},a))) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\text{return } \vec{x}) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\text{return } \vec{x})[\vec{t}/\vec{x}] && \text{(Lemma 9)} \\
&\equiv (\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}) \\
&\equiv (\vec{t},a)
\end{aligned}
$$

- $e \equiv f^{\mathrm{sid}}(\vec{t}, a)$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(f^{\mathrm{sid}}(\vec{t},a))) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(\vec{t})\ \lambda\vec{x}.\ \mathbf{F}_0(f^{\mathrm{sid}})(\vec{x})) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\mathbf{F}_0(f^{\mathrm{sid}})(\vec{x}))[\vec{t}/\vec{x}] && \text{(Lemma 9)} \\
&\equiv (\mathbf{G}_0(\mathbf{F}_0(f^{\mathrm{sid}}))(\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}) \\
&\equiv f^{\mathrm{sid}}(\vec{x},a)[\vec{t}/\vec{x}] && \text{(def. of } \mathbf{G}_0) \\
&\equiv f^{\mathrm{sid}}(\vec{t},a)
\end{aligned}
$$

- $e \equiv \text{if } t \text{ then } e_1 \text{ else } e_2$.

$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(\text{if } t \text{ then } e_1 \text{ else } e_2)) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(t)\ \lambda x.\ \text{if } x \text{ then } \mathbf{F}(e_1) \text{ else } \mathbf{F}(e_2)) && \text{(def. of } \mathbf{F}) \\
&\equiv \mathbf{G}(\text{if } x \text{ then } \mathbf{F}(e_1) \text{ else } \mathbf{F}(e_2))[t/x] && \text{(Lemma 9)} \\
&\equiv (\text{if } x \text{ then } \mathbf{G}(\mathbf{F}(e_1)) \text{ else } \mathbf{G}(\mathbf{F}(e_2)))[t/x] && \text{(def. of } \mathbf{G}) \\
&\equiv (\text{if } x \text{ then } e_1 \text{ else } e_2)[t/x] && \text{(I.H.)} \\
&\equiv \text{if } t \text{ then } e_1 \text{ else } e_2 && \text{(Since } x \notin \mathrm{FV}(e_1) \text{ and } x \notin \mathrm{FV}(e_2))
\end{aligned}
$$

- $e \equiv \text{let } (\vec{x}, a) = e_1 \text{ in } e_2$.

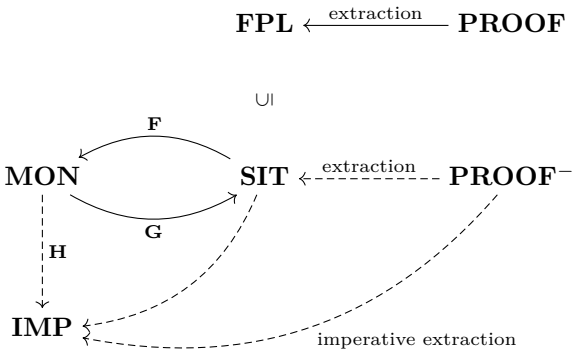
$$
\begin{aligned}
&\mathbf{G}(\mathbf{F}(\text{let } (\vec{x}, a) = e_1 \text{ in } e_2)) \\
&\equiv \mathbf{G}(\text{bind } \mathbf{F}(e_1)\ \lambda \vec{x}.\mathbf{F}(e_2)) && \text{(def. of } \mathbf{F}) \\
&\equiv \text{let } (\vec{x}, a) = \mathbf{G}(\mathbf{F}(e_1)) \text{ in } \mathbf{G}(\mathbf{F}(e_2)) && \text{(def. of } \mathbf{G}) \\
&\equiv \text{let } (\vec{x}, a) = e_1 \text{ in } e_2 && \text{(I.H.)}
\end{aligned}
$$

◀

Since we understand the translation **G** as the definition of the semantics of monadic expressions this theorem states that the transformation **F** is a semantic preserving translation.

7 Conclusion and Further Work

We started by extracting a program from a formal proof of the Quicksort algorithm using the interactive theorem prover Minlog. We then observed, using Monads, that the extracted program behaves imperatively. Using this example for inspiration we defined a restricted functional language **SIT**, a monadic language **MON** and an automatic translation between the two that would allow for fully automatic imperative program extraction. At present it is by chance that the program extracted from the proof behaves imperatively, but we plan to develop a restricted proof calculus which *only* yields imperative programs. The situation is depicted in the diagram below:



The full arrows correspond to the results of this paper whereas the dashed arrows hint at further work:

- a restricted proof calculus, called **PROOF**$^-$, where the extracted programs from proofs are always in the **SIT** language;
- a translation **H** from **MON** to an imperative language **IMP** which would enable, through composition of **F** and **H**, an automatic extraction of imperative programs;
- a direct *imperative extraction* from **PROOF**$^-$ to **H**.

In this paper we considered only first-order constructs, but, ideally, we would like to develop a proof calculus and a program extraction process which combines higher order constructs with imperative features.

References

1 J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

2 Agda. `http://wiki.portal.chalmers.se/agda/`.

3 H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In *Automated Deduction*, volume II of *Applied Logic Series*, pages 41–71. Kluwer, 1998.

4 U. Berger, K. Miyamoto, H. Schwichtenberg, and M. Seisenberger. Minlog – A Tool for Program Extraction Supporting Algebras and Coalgebras. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO 2011*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2011.

5 U. Berger and M. Seisenberger. Proofs, Programs, Processes. *Theory of Computing Systems*, 51(3):313–329, 2012.

6 S. Berghofer. Program extraction in simply-typed higher order logic. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2003.

7 B. Brock, M. Kaufmann, and J. S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293. Springer-Verlag, 1996.

8 C. M. Chuang. *Extraction of Programs for Exact Real Number Computation Using Agda*. PhD thesis, Swansea University, Wales, 2011.

9 The Coq Proof Assistant. `http://coq.inria.fr/`.

10 M. Erwig and D. Ren. Monadification of Functional Programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

11 S. Hallerstede and M. Leuschel. Experiments in program verification using Event-B. *Formal Aspects of Computing*, 24:97–125, 2012.

12 C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

13 Isabelle. `http://isabelle.in.tum.de/`.

14 S. C. Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

15 G. Kreisel. Interpretation of Analysis by means of Constructive Functionals of Finite Types. *Constructivity in Mathematics*, pages 101–128, 1959.

16 J. Krivine. Realizability Algebras: A Program to Well Order $\mathbb{R}$. *Logical Methods in Computer Science*, 7:1–47, 2011.

17 P. Letouzey. A New Extraction for Coq. In *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219, 2003.

18 The Minlog System. `http://www.minlog-system.de`.

19 A. Miquel. Classical Program Extraction in the Calculus of Constructions. In *CSL 2007*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327, 2007.

20 K. Miyamoto, F. Nordvall Forsberg, and H. Schwichtenberg. Program Extraction from Nested Definitions. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 370 – 385. Springer, 2013.

21 E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

22 PRL Project. `http://www.nuprl.org/`.

23 I. Poernomo, J. N. Crossley, and M. Wirsing. *Adapting Proofs-as-Programs: The Curry-Howard Protocol*. Springer, 2005.

24 Swansea Minlog Repository. `http://cs.swan.ac.uk/minlog/`.

25 D. Ratiu and T. Trifonov. Exploring the Computational Content of the Infinite Pigeonhole Principle. *Journal of Logic and Computation*, 22(2):329–350, 2012.

26 S. Ray and R. Sumners. Verification of an In-place Quicksort in ACL2. In D. Borrione, M. Kaufmann, and J. S. Moore, editors, 3*rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2002)*, pages 204–212, Grenoble, 2002.

27 H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Assoc. Symb. Logic and Cambridge Univ. Press, 2012.

28 TIOBE. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.