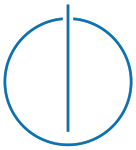
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Arrays in Isabelle

Balazs David Toth



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Arrays in Isabelle

Arrays in Isabelle

Author:	Balazs David Toth
Supervisor:	Prof. Tobias Nipkow
Advisor:	Dr. Peter Lammich
Submission Date:	15.10.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.10.2022 Balazs David Toth

Acknowledgments

I'm extremely grateful for the excellent support of my advisor Dr. Peter Lammich and my supervisor Prof. Tobias Nipkow.

Abstract

Functional lists are a convenient and easy-to-use data structure in functional programming. However, compared to imperative arrays, `lookup` and `update` operations on lists have a significantly worse runtime of $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$. For this reason, we created an automatic refinement of linearly used lists to arrays and also non-linearly used lists to diff arrays. Diff arrays are persistent and store their updates in a tree-like structure next to a plain array. With them, we can have lookup and update operations in $\mathcal{O}(1)$ for its most recent version.

Parallel to the refinement, we will automatically create equivalence proofs using the Isabelle/HOL theorem prover and its Imperative/HOL and separation logic facilities to ensure that the two versions of the program are doing the same.

Contents

1 Introduction

Functional lists (Figure 1.1) are a convenient and easy-to-use data structure in functional programming. However, compared to imperative arrays, `lookup` and `update` operations on lists have a significantly worse runtime of $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$.

```
datatype 'a list = Nil | Cons 'a "'a list"

fun lookup :: "'a list ⇒ nat ⇒ 'a option" where
  "lookup Nil _ = None"
| "lookup (Cons x _) 0 = Some x"
| "lookup (Cons _ xs) (Suc n) = lookup xs n"

fun update :: "'a list ⇒ nat ⇒ 'a ⇒ 'a list" where
  "update (Cons _ xs) 0 y = Cons y xs"
| "update (Cons x xs) (Suc n) y = Cons x (update xs n y)"
| "update xs _ _ = xs"
```

Figure 1.1: An example implementation of a functional list.

Then why not replace lists with arrays at compile time? Like that, we can preserve the functional language's methodological benefits and the runtime of imperative arrays. In section 5.10, we will apply this simple method. But the problem with the method is that arrays are ephemeral, meaning that updates are destructive, like for most imperative data structures (Okasaki 1998, p.2). Contrary to imperative languages, all data structures are automatically persistent in functional languages (ibid., p.2). If we use a list in a non-linear way, we would need to copy the corresponding array, which is costly in terms of runtime and memory.

To cope with that, we use diff arrays (chapter 4), also called trailer arrays (Bloss 1989), which store their updates in a tree-like structure next to the array (Kumar, Blelloch, and Harper 2017, p.706). With them, we can have lookup and update operations in $\mathcal{O}(1)$ for the most recent version of the data structure.

This thesis aims to automatically convert functional programs using lists to imperative

ones using (diff) arrays. Additionally, we will automatically create equivalence proofs to ensure that the two versions of the program are doing the same. We use the Isabelle/HOL theorem prover (chapter 2) for this purpose. We can reason about imperative programs using Isabelle's Imperative/HOL framework (section 2.1) and a separation logic (chapter 3) built on top of it. Using this setup and a verified diff array implementation (chapter 4), we implemented an infrastructure to parallelly translate functional programs to imperative ones and create equivalence proofs (chapter 5). The full source code is in Toth 2022. Concluding, we refined an example algorithm using our developed tool (chapter 6).

2 Isabelle/HOL

Isabelle is a theorem prover implemented in Standard ML, which supports multiple logical systems. We use Isabelle/HOL (Higher-Order Logic), the most used logical system, and the Isabelle/Isar framework, which provides a structured proof language (Wenzel et al. 2004). For detailed descriptions of Isabelle, we refer to ibid. and Nipkow 2013.

2.1 Imperative/HOL

Imperative/HOL is a framework built on Isabelle/HOL for reasoning about imperative programs and data structures. It works with a polymorphic heap model and a "Haskell-like" state-exception monad (Launchbury and Jones 1995). The monadic setup allows straightforward implementations of programs in do-notation using both imperative and functional language constructs. It also supports efficient code generation for various target languages like SML, OCaml, Haskell, and Scala. Additionally, already existing proof automation tools of Isabelle/HOL are easily reusable (Bulwahn, Krauss, Haftmann, et al. 2008, p.134). The heap consists of two mappings, which describe references to values and arrays, and a counter which bounds the currently used address space (ibid., p.140). This heap is the state of the state-exception monad, in which imperative programs can be represented (ibid., p.140 f.). Like that, an imperative program with the return type α is encoded as a value of type α *Heap* (ibid., p.141).

Figure 2.1 is an example function using do-notation for the simple addition of two natural numbers stored on the heap. The program first dereferences the two pointers and then returns their sum. Because imperative programs inside the monad are not required to

```
partial_function (heap) example :: "nat ref ⇒ nat ref ⇒ nat Heap" where
  "example a b = do {
     a ← !a;
     b ← !b;
     return (a + b)
  }"
```

Figure 2.1: An example of an imperative function in Imperative/HOL.

terminate and can raise an exception at any point, we need to use `partial_functions` (Wenzel et al. 2004, p.263) which support non-termination contrary to the regular `funs` or `functions`.

3 Separation Logic

When reasoning about programs that mutate memory, we need to keep track of which parts of the memory can get mutated by which part of the program. Separation logic copes with that by representing satisfiability by a heap and having a conjunction operator that states that the heap can be split into two disjoint parts (Calcagno, O'Hearn, and Yang 2007). Separation logic extends Hoare's logic, such that it is possible to reason about local mutable data structures (Reynolds 2002).

3.1 Separation Logic in Imperative/HOL

The separation logic implementation of Imperative/HOL (Lammich and Meis 2012) is formalized following Calcagno, O'Hearn, and Yang 2007 (Lammich 2017). Assertions are the clauses of the separation logic and describe if a partial heap[1] satisfies a predciate and a well-formedness condition (ibid., p.482).
The three constant atomic assertions are `true`, `false` and `emp`: All heaps satisfy `true`, no heap satisfies `false` and just the empty heap satisfies `emp` (ibid., p.482). Moreover, $\texttt{p}\rightarrow_r\texttt{v}$, $\texttt{p}\rightarrow_a\texttt{xs}$, and $\uparrow\texttt{b}$ are the remaining (non-constant) atomic assertions (ibid.). $\texttt{p}\rightarrow_r\texttt{v}$ describes a heap, where value `v` is at address `p` (ibid.). Analogously, $\texttt{p}\rightarrow_a\texttt{a}$ is a heap where an array `a` holds the same values as the list `xs` and is at the address `p` (ibid.). Furthermore, we can model additional conditions using the pure assertion $\uparrow\texttt{b}$ (ibid.). It describes an empty heap if the boolean clause `b` is true and no heap otherwise (ibid.).
The Imperative/HOL separation logic lifts the standard boolean connectives to assertions, such that they form a boolean algebra themselves (ibid.). Doing so, the separation

[1] A heap, which is restricted to a specific set of adresses

conjunction `P * Q` means that the heap consists of two disjoint parts, where one satisfies `P` and the other `Q` (Lammich 2017, p.483). Additionally, there are also has lifted versions of the universal and existential quantifiers and entailment (ibid.).
The library builds a powerful proof automation for Hoare triples ($\langle$`P`$\rangle$ `c` $\langle$`Q`$\rangle$[2]) with assertions as pre- and postconditions on top of that (ibid.). Its main method is called `sep_auto` and combines among other things simplification and application of previously defined separation logic rules (ibid.). If the method gets stuck, it shows its current state, such that it is possible to continue the proof manually (ibid.). For a more detailed description we refer to ibid.

[2]Read: "If precondtition P is satisfied and program c runs, then its result fulfills the postcondition Q". When reasoning about garbage-collected languages, $\langle P\rangle c\langle Q\rangle_t$ can be used as short form of $\langle P\rangle c\langle \lambda x. Qx * true\rangle$

4 Verified Diff Array Implementation

In the following, we will implement and verify diff arrays using Imperative/HOL (section 2.1) and separation logic (section 3.1) facilities.

4.1 Cell

As the base building block for our diff array implementation, we use a sum type called `cell` (Figure 4.1).

```
datatype 'a::"countable" cell = Array "'a array" | Upd nat "'a" "'a cell ref"
```

Figure 4.1: Cell type

It either directly contains an array or an update at a specific index and a reference to another `cell` stored on the heap. The type of the `cell`'s value must be `countable` because the heap can only store `countable` types. Through that, we can derive that `cell` as a whole is `countable` and satisfies the `heap` type class. As an abstract representation of `cell`, we implement `cell'` (Figure 4.2), which only differs from `cell` in that it stores a list instead of an array.

```
datatype 'a::"countable" cell' = Array' "'a list" | Upd' nat "'a" "'a cell ref"
```

Figure 4.2: Cell' type

Using separation logic assertions, we use the cell assertion function in Figure 4.3 to correspond cells with their abstractions.
The abstraction happens in the first case of the function, such that we assert that the array of `cell` points to the specified list in `cell'`, meaning that the array contains the same elements as the list. For the update cases, we simply assert that the index, value,

```
fun cell_assn where
  "cell_assn (Array' xs) (Array a) = a ↦a xs"
| "cell_assn (Upd' i' val' p') (Upd i val p) = ↑(i = i' ∧ val = val' ∧ p = p')"
| "cell_assn _ _ = false"
```

Figure 4.3: Cell assertion

and pointer are the same using a pure assertion. One would naturally think of recursing on the pointer here, but it is not possible due to how separation logic works: Since we can have multiple updates which point to the same array, we would recurse multiple times to the leaf node. It would result in separating multiple assertions representing the same pointer, which yields `false` in separation logic. We would restrict ourselves to one version per diff array, which defeats the purpose of diff arrays.

4.2 Diff Array Relation

We solve this issue by having references of all `cells` of a diff array, and their corresponding `cell`'s in a list. Using this list, which we will mostly call `t`, we can assert the structure of the diff arrays using a pure assertion, meaning standard boolean logic. We create the relation (Figure 4.4) by recursing on the number of stored updates before reaching the actual array.

```
fun diff_arr_rel' ("(_ ⊢ _ ~⇩_ _)" [51, 51, 51, 51] 50)  where
  "diff_arr_rel' t xs 0 a ⟷ (a, Array' xs) ∈L t"
| "diff_arr_rel' t xs (Suc n) a ⟷ (∃i x a' xs'.
      (a, Upd' i x a') ∈L t
    ∧ diff_arr_rel' t xs' n a'
    ∧ xs = xs'[i:=x]
    ∧ i < length xs'
)"
```

Figure 4.4: Diff array relation[1]

In the base case, when the number of updates is zero, we assert that `t` contains the abstract list representation and the `cell` reference. On the other hand, if the number of updates exceeds zero, we assume that the `cell` reference is in `t` together with an update entry. Furthermore, the index of the update needs to be in bounds of the list abstraction, and the whole relation needs to hold recursively for the reference in the

[1] $x \in_L xs$ is a short-hand notation for $x \in set\ xs$

update with the update reversely applied. As a short-hand notation for the diff array relation, we use $t \vdash xs \sim_n a$. In the following, we mostly will not care about the exact number of updates and existentially quantify it with the definition in Figure 4.5 and its corresponding short-hand notation $t \vdash xs \sim a$.

```
definition diff_arr_rel ("(_ ⊢ _ ~ _)" [51, 51, 51] 50) where
  "diff_arr_rel t xs a ≡ ∃n. t ⊢ xs ~⇩n a"
```

Figure 4.5: Existentially quantified diff array relation

A straightforward property we can prove at this point is that we can add arbitrary elements to the list `t` without interfering with the relation (Figure 4.6).

```
lemma diff_arr_rel_cons: "t ⊢ xs ~ diff_arr ⟹ x # t ⊢ xs ~ diff_arr"
```

Figure 4.6: Adding an element to a diff array relation does not interfere with it

4.3 Master Assertion

To build the bridge between the list on which the diff array relation is working and the cell assertions, we define a master assertion (Figure 4.7).

```
definition master_assn :: "('a cell ref * 'a::heap cell') list ⇒ assn" where
  "master_assn t = fold_assn (map (λ(p, c'). ∃A c. p ↦r c * cell_assn c' c) t)"
```

Figure 4.7: Master assertion

For that, we map `t` to assertions by firstly dereferencing the `cells` using existential quantification in separation logic and the `points-to` relation for references. Secondly, we relate this `cell` with the corresponding `cell'` using the cell assertion. In the next step, we fold the assertions using the separation conjunction. The implementation of this `fold` is straightforward and uses the empty assertion as the start value (Figure 4.8). We show some basic properties of this fold to make the reasoning over the master assertion easier. They do not yield surprises but let us create some useful lemmas for "opening" and "closing" master assertions (Figure 4.9).[2]

[2]With "open", we mean extracting an element from the master assertion, and with "close", adding an element to the master assertion.

```
definition fold_assn :: "assn list ⇒ assn" where
  "fold_assn assns = foldr (*) assns emp"
```

Figure 4.8: Fold assertions

```
lemma open_master_assn':
  assumes "(p, c') ∈L t"
  shows "master_assn t =
          (∃A c. p ↦r c * cell_assn c' c) * master_assn (remove1 (p, c') t)"

lemma open_master_assn:
  assumes "(p, c') ∈L t"
  shows "master_assn t
           ⟹A (∃A c. p ↦r c * cell_assn c' c) * master_assn (remove1 (p, c') t)"

lemma close_master_assn_array: "(a, Array' xs) ∈L t
  ⟹ a' ↦a xs * a ↦r cell.Array a' * master_assn (remove1 (a, Array' xs) t)
      ⟹A master_assn t"

lemma close_master_assn_upd: "(a, Upd' i x a') ∈L t
     ⟹ a ↦r Upd i x a' * master_assn (remove1 (a, Upd' i x a') t) ⟹A master_assn t"

lemma close_master_assn_upd': "(a, Upd' i x a') ∈L t
     ⟹ a ↦r Upd i x a' * master_assn (remove1 (a, Upd' i x a') t) = master_assn t"
```

Figure 4.9: Opening and closing master assertions

We will use these lemmas extensively to verify the diff array operations by opening and closing the master assertion using the diff array relation.
Another important property of master assertions is that all `cell` references in `t` must be distinct (Figure 4.10). If that is not the case, two assertions with the same pointer would be separated, which in separation logic always results in `false`.

```
lemma master_assn_distinct: "h ⊨ master_assn t ⟹ distinct (map fst t)"

lemma master_assn_distinct': "master_assn t ⟹A master_assn t * ↑(distinct (map fst t))"
```

Figure 4.10: Pointers in a master assertion are distinct

4.4 Diff Array Operations

In the following, we will use references to cells as diff arrays and create a type synonym for that (Figure 4.11).

```
type_synonym 'a diff_arr = "'a cell ref"
```

Figure 4.11: Diff array type synonym

```
qualified definition from_array ::
  "('a::heap) array ⇒ 'a diff_arr Heap"
where
  "from_array a = do {
    ref (Array a)
  }"

qualified definition from_list ::
  "('a::heap) list ⇒ 'a diff_arr Heap"
where
  "from_list xs = do {
    a ← Array.of_list xs;
    from_array a
  }"
```

Figure 4.12: Diff array constructors

4.4.1 Constructors

For creating diff arrays from plain arrays and lists, we define two constructor functions (Figure 4.12). The latter creates a plain array of the list and then uses the former, which simply creates an `Array`-cell and then returns a reference to it.

To verify imperative implementations, we use Hoare triples, as introduced in chapter 3. When creating a new diff array, a new master assertion, as well as a new diff array relation, are introduced in the postcondition of the Hoare triple. We have two versions of the proofs (Figure 4.13), once with concrete `ts` and once existentially quantified over `t`, because later, we mostly will not care about the concrete `ts` anymore. The preconditions of the list proofs are empty, meaning that the list constructors can be called everywhere. The array proofs have an array pointer assertion as their precondition, which is naturally introduced when arrays are created.

The proofs are all based on `from_array'`, which we prove by providing instances for the existential quantifications of the master assertion and the diff array relation. The unfolded master assertion says that the reference returned by the constructor function needs to point to an `Array` cell and that the array inside the cell needs to point to the provided list. The implementation of the constructor function directly fulfils these two requirements. To also fulfill the diff array relation, we instantiate the length of the update list with zero since we have no updates yet. In its base case, the diff array

```
lemma from_array' [sep_heap_rules]:
  "<a ↦a xs>
      from_array a
   <λr. let t = [(r, Array' xs)]
        in master_assn t * ↑(t ⊢ xs ~ r)>"

lemma from_list' [sep_heap_rules]:
  "<emp>
    from_list xs
   <λr. let t = [(r, Array' xs)]
        in  master_assn t * ↑(t ⊢ xs ~ r)>"

lemma from_array [sep_heap_rules]:
  "<a ↦a xs> from_array a <λr. ∃At. master_assn t * ↑(t ⊢ xs ~ r)>"

lemma from_list [sep_heap_rules]:
  "<emp> from_list xs <λr. ∃At. master_assn t * ↑(t ⊢ xs ~ r)>"
```

Figure 4.13: Diff array constructor proofs

relation requires the cell and its list abstraction to be in `t`, which is true because `t` is the singleton list containing precisely this pair of values.

4.4.2 Lookup

After constructing a diff array, we also want to be able to look up values in it (Figure 4.14). For that, we first check if we are at a leaf node. If yes, we look the value up in the underlying array. Otherwise, we compare the index of the update we are looking at with the index of the value we want to look up. In case they equal, we can directly return the value of the update, or else we recurse on the diff array to which the update is pointing.

```
qualified partial_function (heap) lookup ::
  "('a::heap) diff_arr ⇒ nat ⇒ 'a Heap"
where
  "lookup diff_arr i = do {
      cell ← !diff_arr;
      case cell of
        Array array             ⇒
          Array.nth array i
      | Upd i' value diff_arr' ⇒
          if i = i'
          then return value
          else lookup diff_arr' i
  }"
```

Figure 4.14: Diff array lookup

To prove the implementation correct (Figure 4.15), we assume a master assertion, a diff array, and that the index we want to look up is in the array's bounds. After a function execution, the same master assertion should hold, and the result of the function should

yield the same value as the list abstraction at that index.

```
lemma lookup [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ diff_arr ∧ i < List.length xs)>
     lookup diff_arr i
   <λr. master_assn t * ↑(r = xs!i)>"
```

Figure 4.15: Diff array lookup proof

The proof is an induction on the number of updates of the diff array. After opening the master assertion for the current diff array cell, the separation logic automation can prove both cases of the induction.

4.4.3 Update

For updating diff arrays (Figure 4.16), we follow the heuristic of Bloss 1989, p.27, that the most recent version of an array is mainly accessed. Consequentially, when updating an array without an update tree, we update destructively and create a new pointer as the most recent version. The previous version gets updated by assigning it an update cell containing the reverted update and a pointer to the new version. Following the same schema, we realize arrays, which have an update tree and then update the newly created arrays destructively.

```
qualified partial_function (heap) update ::
    "('a::heap) diff_arr ⇒ nat ⇒ 'a::heap ⇒ 'a diff_arr Heap"
where
  "update diff_arr i v = do {
      cell ← !diff_arr;
      case cell of
        Array arr ⇒ do {
          new_diff_arr ← ref (Array arr);
          old_v ← Array.nth arr i;
          diff_arr :=R Upd i old_v new_diff_arr;
          Array.upd i v arr;
          return new_diff_arr
        }
      | Upd _ _ _ ⇒ do {
          arr ← realize diff_arr;
          Array.upd i v arr;
          ref (Array arr)
        }
  }"
```

Figure 4.16: Diff array update

By realizing, we mean creating a new array with the updates of the update tree applied. Like this, we can guarantee that accesses to the most recent versions always have a

runtime of $\mathcal{O}(1)$.

The utilized `realize` function (Figure 4.17) recurses the update tree down to the plain array, copies it, and applies the updates of the update tree destructively to the copy.

```
qualified partial_function (heap) realize ::
  "('a::heap) diff_arr ⇒ 'a array Heap"
where
  "realize diff_arr = do {
    cell ← !diff_arr;
     case cell of
        Array arr ⇒ do {
            len ← Array.len arr;
            xs  ← Array.freeze arr;
            Array.make len (List.nth xs)
        }
      | Upd i v diff_arr ⇒ do {
          arr ← realize diff_arr;
          Array.upd i v arr
        }
  }"
```

Figure 4.17: Realize diff array

We prove `realize` again by induction on the number of updates of the diff array (Figure 4.18). To resolve the case distinction on the `cell` type, we can, in the base case as well as the induction step, open the master assertion. After knowing what kind of cell we are looking at currently, we can close the master assertion again and the separation logic automation can prove that the function preserves the master assertion and additionally creates a plain array that contains the same elements as the diff array.

```
lemma realize [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ diff_arr)>
     realize diff_arr
   <λarr. master_assn t * arr ↦a xs>"
```

Figure 4.18: Diff array realize proof

In order to prove the update operation correct, we first prove an auxiliary lemma (Figure 4.19). It states that a diff array relation still holds if we replace a leaf node with another one that differs in one entry and an update that reverts this difference.

An additional assumption is that the `t` under which the diff array relations hold only contains distinct cells. In section 4.3, we have already shown that a master assertion yields this property.

We can prove the lemma using a structural induction on the diff array relation. In

```
lemma update_diff_arr_rel: "⟦
  i < List.length xs;
  (diff_arr, Array' xs) ∈ₗ t;
  distinct (map fst t);
  t ⊢ xs' ~⇩n' diff_arr'
⟧ ⟹ ∃n. (new_diff_arr, Array' (xs[i := v])) #
         (diff_arr, Upd' i (xs ! i) new_diff_arr) #
         remove1 (diff_arr, Array' xs) t ⊢ xs' ~⇩n diff_arr'"
```

Figure 4.19: Update diff array relation

the base case, the number of updates that the diff array contains is zero. We can now differentiate two cases: The replaced diff array is the same as the one for which the diff array relation holds or not. We know that the number of updates of the diff array in the diff array relation increases in the first case to one and stays zero in the second. We provide these as the witnesses of the existentially quantified number of updates for the conclusion. The proof automation will then prove the rest.

In the induction step of the structural induction, we can unfold one step of the diff array relation. Using the induction hypothesis, we can now show the proposition for the underlying updated array with a fixed number of updates. Finally, we can show the overall conclusion by witnessing the number of updates with the just fixed number incremented by one and using the definition of the diff array relation. This lemma is one of the main building blocks for proving the update operation correct.

```
lemma update [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ diff_arr ∧ i < List.length xs)>
     update diff_arr i v
   <λdiff_arr. ∃ₐt'. master_assn t' *
    ↑((∀xs' diff_arr'. t ⊢ xs' ~ diff_arr' ⟶ t' ⊢ xs' ~ diff_arr') ∧
      (t' ⊢ xs[i := v] ~ diff_arr))>"
```

Figure 4.20: Diff array update proof

The lemma of the update operation has the same preconditions as the lookup operation, but its postconditions are slightly more complicated (Figure 4.20). Firstly, there is a new `t'` because the cells of the diff array are not staying the same. The master assertion still needs to hold for it, but the diff array relation now relates the diff array with the updated list. Finally, it is also essential that all the diff array relations that were holding for the `t` of the precondition also hold for the new `t'`. Otherwise, we would lose the old versions of the diff array.

To prove this lemma, we differentiate whether the diff array's number of updates is zero. That corresponds to the case distinction on the cell in the function implementation.
The easier case[3] is when the number of updates is not zero, and consequently, the cell is an update cell. Here, we realize the diff array to a new array and then update it. Correspondingly, the proof uses our previous proof for `realize` (Figure 4.18) and the Imperative/HOL-library proof for destructive array updates. We now know that the master assertion still holds for `t` and, additionally, that there is a new diff array that relates to the list updated at index `i` with the value `v`. As the last step, we can now construct the new `t'` by appending the new diff array to the previous `t` and using the previously shown rules for appending to master assertions (Figure 4.9).
For the case that there are no update entries, we first introduce the additional premise that the cells in `t` are unique using `master_assn_distinct` (Figure 4.10). Based on that, we can later apply `update_diff_arr_rel` (Figure 4.19). Next, we can open the master assertion and run the separation logic automation. It stops at the point where we need a witness for the new `t'`. We can construct it by replacing the array cell with the array cell of the new version and an update cell, which reverts the new change also to keep the old version. Finally, we can apply `update_diff_arr_rel` and provide zero as the number of updates for the new updated entry to conclude the proof.

4.4.4 Length

Since it can also be helpful to know the length of an array, we implement an operation for that, too (Figure 4.21).

```
qualified partial_function (heap) length ::
  "('a::heap) diff_arr ⇒ nat Heap"
where
  "length diff_arr = do {
      cell ← !diff_arr;
      case cell of
        Array array   ⇒ Array.len array
      | Upd _ _ diff_arr ⇒ length diff_arr
  }"
```

Figure 4.21: Diff array length

[3]Because we have already proofs for `realize` (Figure 4.18).

The implementation is straightforward and simply recurses the update chain down to the array cell and returns the length of the plain array[4]. The proof is similarly straightforward and can be done, for example, by structural induction on the diff array relation of the precondition (Figure 4.22). Both cases of the induction can be proven analogously by first opening and later closing the master assertion, accompanied by the separation logic automation.

```
lemma length [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ diff_arr)>
     length diff_arr
   <λlen. master_assn t * ↑(len = List.length xs)>"
```

Figure 4.22: Diff array length proof

4.4.5 Safe Operations

Before we can go on to automatically replace lists with arrays, we need to simplify one detail of the implementations of the lookup and update operations. The operations expect that the provided index is inside the bounds of the array, and it would be cumbersome just to allow the translation of functions accounting for it.

To solve this issue, we define a lookup operation that is undefined for indices out of bounds in a similar way as the standard `List.nth` operation (Figure 4.23). Like that, the program will exceptionally terminate on an index out of bounds.

The update operation does not even need that (Figure 4.24). We simply do not update anything if the index is out of bounds.

The proofs for both operations use separation logic proof automation with our previous proofs of the according operations (Figure 4.25).

```
qualified definition lookup where
  "lookup arr i = do {
    len ← Diff_Arr.length arr;
    if i < len
    then Diff_Arr.lookup arr i
    else return (undefined(i - len))
  }"
```

Figure 4.23: Safe diff array lookup

[4]As a future improvement of the runtime, one could also store the length directly into the cells.

```
qualified definition update where
  "update arr i v = do {
    len ← Diff_Arr.length arr;
    if i < len
    then Diff_Arr.update arr i v
    else return arr
  }"
```

Figure 4.24: Safe diff array update

```
lemma lookup_safe [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ a)>
     lookup a i
  <λr. master_assn t * ↑(r = xs!i)>"

lemma update_safe [sep_heap_rules]:
  "<master_assn t * ↑(t ⊢ xs ~ diff_arr)>
     update diff_arr i v
  <λdiff_arr. ∃ₐt'. master_assn t' *
    ↑((∀xs' diff_arr'. t ⊢ xs' ~ diff_arr' ⟶ t' ⊢ xs' ~ diff_arr')
      ∧ (t' ⊢ xs[i := v] ~ diff_arr))>"
```

Figure 4.25: Safe diff array operation proofs

5 Automatic Refinement

As the central infrastructure to automatically refine lists to (diff) arrays, we define a predicate `hnr Γ c Γ' a` (short for heap-nres refinement; Figure 5.1) similar to Lammich 2017, p.490 and Lammich 2019, p.12. `hnr` relates an abstract function `a` with an imperative program `c` in the Imperative/HOL heap monad. Additionally, Γ describes the heap precondition for `c` using separation logic assertions and Γ′ describes the heap after executing `c`, also using separation logic assertions. In our terms, `hnr` predicates read like this: If `c` runs on a heap that fulfills Γ, then the result of `c` is the same as the result of `a`, and the heap afterward fulfills Γ′. `hnr` is defined using a Hoare triple for garbage-collected languages.

```
definition hnr where
  "hnr Γ c Γ' a = (
     case a of None ⇒ True |
               Some a ⇒ <Γ> c <λr. Γ' a r>t
  )"
```

Figure 5.1: Hnr predicate

`a` is defined using an option type, such that the abstract function cannot just be total but also partial. It will help us automate the translation of recursive functions (section 5.8). Consequently, we can directly prove that `hnr` holds for failed or non-terminating functions (Figure 5.2).

```
lemma hnr_none [simp]: "hnr Γ c Γ' None"
```

Figure 5.2: Failing hnr

While building up the refinement infrastructure, we assume input functions with a termination proof that are monadified in the option monad similar to Wimmer, Hu, and Nipkow 2018. That means that every value and constant is assigned to a separate

variable using monadic binds or lets. Furthermore, recursive functions are assumed to use the option monad fixed-point operator (Krauss 2010, p.5). Unfortunately, an automatic translation of functions into the assumed format is not implemented yet (see chapter 7).

```
definition example_1 where
  "example_1 xs = do {
    let c1 = List.length xs;
    let c2 = 2;
    let c3 = c1 < c2;
    if c3 then do {
      let c4 = 1;
      let c5 = xs[c4 := c4];
      Some c5
    } else do {
      let c6 = 2;
      let c7 = xs[c6 := c6];
      Some c7
    }
  }"
```

Figure 5.3: Example of a monadified function

We will collect all the `hnr` rules inside a rule set called `hnr_rule` and create conversions between `hnr` and Hoare triples to prove these rules (Figure 5.4).
Further, we will build up the `hnr` rules such that every value has an assertion. The basic assertion, therefore, is the identity assertion, which states that a value refines to itself. For that, we wrap the equality operator inside a definition so that no accidental simplifications can happen.

```
named_theorems hnr_rule

lemma hnr_hoare: "(∀x. a = Some x ⟶ <Γ> c <λr. Γ' x r>ₜ) ⟷ (hnr Γ c Γ' a)"

lemmas hnrI = hnr_hoare[THEN iffD1, rule_format]
lemmas hnrD = hnr_hoare[THEN iffD2, rule_format]

definition id_rel where "id_rel a c ≡ c = a"

abbreviation id_assn where "id_assn a c ≡ ↑(id_rel a c)"

abbreviation array_assn where "array_assn xs xsi ≡ xsi ↦ₐ xs"

lemma hnr_post_cons:
  assumes
    "hnr Γ fi Γ' f"
    "⋀x xi. Γ' x xi ⟹ₐ (Γ'' x xi)"
  shows
    "hnr Γ fi  Γ'' f"
```

Figure 5.4: Basic hnr setup

5.1 Constant Rule and Pass Rule

To cope with our self-set rule that every value gets an assertion, we introduce a `hnr` rule, which creates identity assertions for constants (Figure 5.5).

```
lemma hnr_const: "hnr Γ (return x) (λr ri. Γ * id_assn r ri) (Some x)"
```

Figure 5.5: Constant rule

If no other `hnr` rule can be applied, this will be our fallback rule. The proof for the rule is a conversion of the `hnr` predicate to a Hoare triple, and then the separation logic proof automation does the rest. In the case that we already have an assertion for a value, we do not need to create a new assertion but can simply pass on the existing one. Figure 5.6 shows a general rule for that purpose. Later, we will specify this rule for the different assertion types, for example, the diff array master assertion. However, in the first step, we just specify it for identity assertions.

```
lemma hnr_pass_general: "hnr (Γ x xi) (return xi) Γ (Some x)"
lemma hnr_pass: "hnr (id_assn x xi) (return xi) id_assn (Some x)"
```

Figure 5.6: Pass rule

Again, the proof consists of a conversion to a Hoare triple and separation logic proof automation.

5.2 Keep - Drop

Since Imperative/HOL translates by default to a garbage-collected language, we need a facility to drop assertions of data structures that go out of scope. In the branches of the `if`-statement in Figure 5.3, for example, the variables `c4 - c7` are going out of scope after the branches are left, such that it would not be possible to keep assertions for them. Therefore, we create the definition in Figure 5.7 to separate the assertions we want to keep and drop in our `hnr` rules.

Γ describes all our current assertions, which we then split into what we want to keep (`K`) and what we want to drop (`D`). For instance, rules for translating case distinctions

```
definition Keep_Drop where
  "Keep_Drop Γ K D ≡ Γ ⟹A K * D"
```

Figure 5.7: Separation of assertions to keep or drop

will produce such goals. To resolve them, we first unfold the definition by using the initialization rule (Figure 5.8).

```
lemma init:
  assumes
    "Γ ⟹A K * D"
  shows
    "Keep_Drop Γ K D"
```

Figure 5.8: Keep - Drop initialization rule

After that, there are three different rules (Figure 5.9) that we try to apply to resolve the goal:

1. We try to split up the assertion as far as possible using the separation conjunction.

2. If we cannot split up the assertion anymore, we try to keep it by replacing its drop part with the empty assertion. Note that we do not just use $\Gamma \Longrightarrow_A \texttt{emp} * \Gamma'$ to allow more sophisticated matching methods than simple entailment. For example, for matching master assertions, we want to allow different orders of elements inside it.

3. If keeping an assertion does not work, we drop it by replacing its keep part with the empty assertion and putting the whole assertion into the drop part.

We can use an Eisbach method (Figure 5.10) to put these three rules together as described. Further, we put the initialization rule in front of it and execute the step as often as possible to resolve keep-drop statements automatically.

```
lemma split:
  assumes
    "Γ1 ⟹A K1 * D1"
    "Γ2 ⟹A K2 * D2"
  shows
    "Γ1 * Γ2 ⟹A (K1 * K2) * (D1 * D2)"

lemma keep:
  assumes
    "Γ ⟹A Γ'"
  shows
    "Γ ⟹A Γ' * emp"

lemma drop: "Γ ⟹A emp * Γ"
```

Figure 5.9: Keep - Drop rules

```
method keep_drop_step methods keep_atom =
  rule split | (rule keep, keep_atom) | rule drop

method keep_drop methods keep_atom =
  rule init, ((keep_drop_step keep_atom)+; fail)
```

Figure 5.10: Keep - Drop methods

5.3 Normalization

For example, after resolving a keep-drop clause, the assertions might not be normalized anymore, meaning that the assertion can contain empty clauses and does not have the default bracketing. We introduce the definition in Figure 5.11 to mark places where this can happen.

```
definition Norm where
  "Norm Γ Γ' ≡ Γ ⟹A Γ'"
```

Figure 5.11: Normalization definition

When reaching such a goal, we can do the normalization by first unfolding the definition. The separation logic library has a rule collection called `star_aci`, which can then normalize the assertion by giving it to the simplifier. It is also possible to hand in custom normalization rules. After that, as the last step, we can solve the entailment by reflexivity.

```
lemma normI: "Γ ⟹A Γ' ⟹ Norm Γ Γ'"

method normalize uses rules =
  rule normI, (simp only: star_aci rules)?; rule ent_refl
```

Figure 5.12: Normalization procedure

5.4 Merge

After an `if`- or `case`-statement, every branch has its own assertion, which we need to merge into one to continue with the function refinement. As for keep-drop and normalization, we introduce a definition again (Figure 5.13).

```
definition Merge where
  "Merge Γa Γb Γc ≡ Γa ∨A Γb ⟹A Γc"
```

Figure 5.13: Merge definition

The first two parameters, Γ_a and Γ_b describe the postconditions of two different branches. We connect them with a separation logic disjunction because either one or the other branch runs. The common elements of the two assertions then come together in Γ_c. Because of our keep drop routine, which we expect to be executed prior to the merge, we can assume that the postconditions of both branches are the same. Since they may not be normalized, we normalize them in the same way as in the previous section and then resolve the merge by reflexivity. In Figure 5.14 we put these steps together using an Eisbach method.

```
lemma merge_refl: "Merge Γ Γ Γ"

method merge uses rules = (simp only: star_aci rules)?, rule merge_refl
```

Figure 5.14: Merge method

5.5 Hnr Rules

In the following, we define some general `hnr` conversion rules of language statements, which we collect in the previously defined rule set called `hnr_rule` (Figure 5.4).

5.5.1 Tuple

When reaching the creation of a tuple, we can translate it into the binding of two already translated programs, which we then combine into a tuple. As a result, we have two `hnr` clauses in the assumption. The first of these has the same precondition as the conclusion. The second takes the postcondition of the first as its precondition since they run one after the other, and the second can depend on the result of the first one. The postcondition of the second clause is then consequently the postcondition of the conclusion, which can depend on the results of both clauses. To satisfy these dependencies of the conclusion, we deconstruct the tuple using the selectors for its first and second elements.

The proof of this rule is a conversion of the `hnrs` to Hoare triples and separation logic proof automation.

```
lemma hnr_tuple [hnr_rule]:
  assumes
    "hnr Γ ai Γa (Some a)"
    "⋀a ai. hnr (Γa a ai * true) bi (Γb a ai) (Some b)"
  shows
    "hnr
      Γ
      (do { ai' ← ai; bi' ← bi; return (ai', bi') })
      (λx xi. Γb (fst x) (fst xi) (snd x) (snd xi))
      (Some (a, b))"
```

Figure 5.15: Hnr tuple rule

5.5.2 Bind

When binding a variable in the option monad (equivalent to let-bindings), we naturally also want to have a bind in the heap monad. Therefore, we create two `hnr` clauses as assumptions to achieve that - one for the bound value and one for the context of the bound value. Additionally, we introduce a keep-drop clause to drop the dependencies on the locally bound variable again. As described in section 5.2, we always need a normalization clause accompanying the keep-drop clause.
The proof of the rule works by converting to Hoare triples again, then using the proof automation, and unfolding and applying the keep-drop and normalization definitions.

```
lemma hnr_bind [hnr_rule]:
  assumes
    "hnr Γ vi Γ1 v"
    "⋀x xi. hnr (Γ1 x xi) (fi xi) (Γ' x xi) (f x)"
    "⋀x xi r ri. Keep_Drop (Γ' x xi r ri) (Γ'' r ri) (Γ1' x xi r ri)"
    "⋀r ri. Norm (Γ'' r ri) (Γ''' r ri)"
  shows
    "hnr Γ (do { x ← vi; fi x }) Γ''' (do { x ← v; f x })"
```

Figure 5.16: Hnr bind rule

5.5.3 If

To convert an `if`-statement, we assume that we have already converted its two branches and have an identity assertion for its condition and the condition's abstraction. Additionally, we need a merge clause, which merges the two postconditions of the branches

to the postcondition of the whole statement.

The proof of the rule consists of unfolding the merge definition, destructuring the `if`-statement, and using rules for disjunctions in separation logic assertions.

```
lemma hnr_if [hnr_rule]:
  assumes
    "hnr (Γ * id_assn c ci) ai Γa a"
    "hnr (Γ * id_assn c ci) bi Γb b"
    "⋀r ri. Merge (Γa r ri) (Γb r ri) (Γc r ri)"
  shows
    "hnr
      (Γ * id_assn c ci)
      (if ci then ai else bi)
      Γc
      (if c then a else b)"
```

Figure 5.17: Hnr if rule

5.5.4 Case

We defined rules for pattern matching on the sum type of the HOL library, natural numbers, and lists. They are very similar, such that we just describe them generally. In the future, we will evaluate if a general version of these rules is possible. Also, we do not yet support the conversion of case distinctions on refined types and types that contain refined types (chapter 7).

A case rule assumes the conversion of all its branches. Further, it associates keep-drop and normalization clauses with the `hnr` assumptions, such that the bound internal values of the destructured constructors, which the branches can depend on, are dropped again. Using these conversions, we can build up the case statement in the imperative program again. Therefore, the value that is pattern matched needs to be identical in both programs, shown by an identity assertion in the precondition of the `hnr` rule. Due to this, we have the deficiency of not being able to convert pattern matches on refined types, which would have different assertions. Like in the `if` rule (Figure 5.17), a last additional assumption merges the postconditions of the branches again.

To prove the rules, we first convert the `hnrs` to Hoare triples, unfold the merge definition, and split up the conclusion into all possible cases of the pattern match using the separation logic proof automation and its `split` option. Now, we have individual goals

```
lemma hnr_case_sum [hnr_rule]:
  assumes
    "⋀s' si'. hnr (Γ * id_assn s si * id_assn s' si') (cli si') (Γa s' si') (cl s')"
    "⋀l' li' ri r. Keep_Drop (Γa l' li' r ri) (Γa' r ri) (Dropa l' li' r ri)"
    "⋀r ri. Norm (Γa' r ri) (Γa'' r ri)"

    "⋀s' si'. hnr (Γ * id_assn s si * id_assn s' si') (cri si') (Γb s' si') (cr s')"
    "⋀r' ri' ri r. Keep_Drop (Γb r' ri' r ri) (Γb' r ri) (Dropb r' ri' r ri)"
    "⋀r ri. Norm (Γb' r ri) (Γb'' r ri)"

    "⋀r ri. Merge (Γa'' r ri) (Γb'' r ri) (Γc r ri)"
  shows
    "hnr
      (Γ * id_assn s si)
      (case si of Inl l ⇒ cli l | Inr r ⇒ cri r)
       Γc
      (case s of Inl l ⇒ cl l | Inr r ⇒ cr r)"

lemma hnr_case_nat[hnr_rule]:
  assumes
    "hnr (Γ * id_assn n ni) ci0 Γa c0"

    "⋀n' ni'. hnr (Γ * id_assn n ni * id_assn n' ni') (ci ni') (Γb n' ni') (c n')"
    "⋀n ni ri r. Keep_Drop (Γb n ni r ri) (Γb' r ri) (Drop n ni r ri)"
    "⋀r ri. Norm (Γb' r ri) (Γb'' r ri)"

    "⋀r ri. Merge (Γa r ri) (Γb'' r ri) (Γc r ri)"
  shows
    "hnr
      (Γ * id_assn n ni)
      (case ni of 0 ⇒ ci0 | Suc n' ⇒ ci n')
      Γc
      (case n of 0 ⇒ c0 | Suc n' ⇒ c n')"

lemma hnr_case_list [hnr_rule]:
  assumes
    "hnr (Γ * id_assn xs xsi) ci0 Γa c0"

    "⋀x' xi' xs' xsi'.
      hnr
        (Γ * id_assn xs xsi * id_assn x' xi' * id_assn xs' xsi')
        (ci xi' xsi')
        (Γb x' xi' xs' xsi')
        (c x' xs')"
    "⋀x xi xs xsi ri r. Keep_Drop (Γb x xi xs xsi r ri) (Γb' r ri) (Drop x xi xs xsi r ri
    "⋀r ri. Norm (Γb' r ri) (Γb'' r ri)"

    "⋀r ri. Merge (Γa r ri) (Γb'' r ri) (Γc r ri)"
  shows
    "hnr
      (Γ * id_assn xs xsi)
      (case xsi of [] ⇒ ci0 | x#xs ⇒ ci x xs)
      Γc
      (case xs of [] ⇒ c0 | x#xs ⇒ c x xs)"
```

Figure 5.18: Hnr case rules

for each case of the pattern match. We can solve these goals by using the translation assumptions of the branches and some standard rules for disjunctions of separation logic assertions. Additionally, the associated keep drop and normalization clauses help the simplifier.

5.6 Fallbacks

The constant rule (Figure 5.5) cannot translate expressions like `let c3 = c1 < c2` in Figure 5.3, because the expression is not fully evaluated, and the translation cannot depend directly on its abstraction. Theoretically, it would be possible to also monadify the less function and then translate it. However, when an expression does not contain a refined value, there is a more straightforward approach.
First, we can reduce such a translation to an equality assumption of the abstract and translated value, following the precondition of the `hnr` statement. Further, we can introduce an identity assertion, like in the constant rule, but we will have to prove the equality assumption in the next step (Figure 5.19).

```
lemma hnr_fallback:
  assumes
    "⋀h. h ⊨ Γ ⟹ c = ci"
  shows
    "hnr Γ (return ci) (λr ri. Γ * id_assn r ri) (Some c)"
```

Figure 5.19: Hnr fallback rule

We can do that by substituting all the identities of the precondition. In our small example, we would substitute `c1` and `c2` with their translations, which is possible because we do not refine natural numbers and have assertions for every value. In the final step, we can resolve the statement with simple reflexivity. We created an Eisbach method in Figure 5.20 for this procedure.

5.7 Frame Inference

The pre- and postconditions of the `hnr` statements during translation might not just contain the assertions needed for the translation. We will call these additional assertions

```
method extract_pre uses rule =
  (determ‹elim mod_starE rule[elim_format]›)?

lemma models_id_assn:"h ⊨ id_assn x xi ⟹ x = xi"

method hnr_fallback =
  rule hnr_fallback,
  extract_pre rule: models_id_assn,
  ((hypsubst, rule refl) | (simp(no_asm_simp) only: ; fail))
```

Figure 5.20: Hnr fallback method

frame since they can be around the currently needed assertions.

Because it would be cumbersome to formulate all the `hnr` rules with this possible *frame* in mind, we create a rule that automatically does so for us (Figure 5.21).

```
lemma hnr_frame:
  assumes
    "Γp ⟹A Γ * F"
    "hnr Γ fi Γ' f"
  shows
    "hnr Γp fi (λr ri. Γ' r ri * F) f"
```

Figure 5.21: Frame rule

It splits up the precondition into the frame and the needed assertions for proving the `hnr` statement. The frame is then passed on unchanged to the postcondition.

The proof of this rule is a translation to Hoare triples and proof automation.

To apply the rule conveniently, we create an attribute called `framed`. We will use this attribute, inter alia, with our previously defined pass rule (Figure 5.6) because it also needs to work if not just one identity assertion is in the precondition. The converted pass rule looks like the following:

$\Gamma_P \Longrightarrow_A$ `id_assn x xi` $*$ `F`

$\Longrightarrow$ `hnr` Γ_P `(return xi)` $(\lambda$ `r ri. id_assn r ri` $*$ `F) (Some x)`

However, how do we differentiate the assertions that belong to the frame and those that are needed by the next rule? Or in other words, how do we resolve the second assumption of the frame rule? We create a frame inference method for this purpose.

The starting point is the assumption of a goal of the following form: `P` $\Longrightarrow_A$ `Q` $*$ `F`.

We want to split up an assertion `P` into `Q`, which the next `hnr` rule needs, and a frame `F`. Therefore, we rotate through the elements in `Q` by changing associativity stepwise from left to right. For each element, we rotate similarly through `P` until we find a

matching assertion, which we then remove from `P`. The assertions which remain in `P` are consequently the frame.

Figure 5.22 shows an example where `P` consists of the assertions `a`, `b`, `c` and `d`, and `Q` contains `a` and `c`. By removing `a` and `c`, only `b` and `d` remain, which are the frame.

$$\begin{aligned} \texttt{a} * \texttt{b} * \texttt{c} * \texttt{d} &\Longrightarrow_A \texttt{a} * \texttt{c} * \texttt{?F} \\ \texttt{b} * \texttt{c} * \texttt{d} &\Longrightarrow_A \texttt{c} * \texttt{?F} \\ \texttt{b} * \texttt{d} &\Longrightarrow_A \texttt{?F} \end{aligned}$$

Figure 5.22: Frame inference example

We put these steps together in a parameterized Eisbach method taking the matching strategy for assertions to allow also more sophisticated methods than simple entailment reflexivity. We will use that for matching the master assertions of our diff arrays.

5.8 Recursion

We now have almost all the parts together, which we need to translate a program in the option monad to one in the heap monad. Only recursions are missing. Since we assume recursive functions using the option monad fixed-point operator (Krauss 2010, p.5), we will reach a goal of the following form:

`hnr (Γ x xi) ?c ?Γ' (option.fixp_fun f x)`

`?c` is the imperative program we want to construct using the fixed-point operator of the heap monad (ibid., p.8). For that, we create and prove the rule in Figure 5.23.

```
lemma hnr_recursion:
  assumes
    mono_option "⋀x. mono_option (λr. f r x)"
  and
    step: "⋀r ri x xi F. (⋀x' xi' F'. hnr (Γ F' x' xi') (ri xi') (Γ' F' x' xi') (r x'))
        ⟹ hnr (Γ F x xi) (fi ri xi) (Γ'' F x xi) (f r x)"
  and
    norm: "⋀F x xi r ri. Norm (Γ'' F x xi r ri) (Γ' F x xi r ri)"
  and
    mono_heap: "⋀x. mono_Heap (λr. fi r x)"
  shows
    "hnr (Γ F x xi) (heap.fixp_fun fi xi) (Γ' F x xi) (option.fixp_fun f x)"
```

Figure 5.23: Hnr recursion rule

It states that we can construct a recursive function by assuming the `hnr` statement for the recursive call and showing that it also holds for the recursive call wrapped into the

function body. Additionally, we need to assume the monotonicity of the original function and the translated function. Later on, we can show these properties automatically by using a tactic of the partial function package.

The application of the recursion rule is not yet fully automated, such that one needs to manually provide the pre- and postconditions (chapter 7). But we created a helper method applying the framed version of the rule and running the frame inference afterward (Figure 5.24). It also deals with unfolding the recursion parameter if it is a tuple. We need this, because the fixed-point operators just support one recursion parameter, such that we have to bundle multiple ones into a tuple. An example for applying the recursion method is in chapter 6, where we translate the Lomuto partitioning algorithm.

```
method hnr_recursion
  for Γ::"'F ⇒ 'x ⇒ 'xi ⇒ assn" and Γ'::"'F ⇒ 'x ⇒ 'xi ⇒ 'r ⇒ 'ri ⇒ assn"
  methods frame_match_atom =
  rule hnr_recursion[where Γ=Γ and Γ'=Γ', framed],
  ((subst tuple_selector_refl, simp only: fst_conv snd_conv)+)?,
  hnr_frame_inference frame_match_atom
```

Figure 5.24: Hnr recursion method

After resolving the function body using our other `hnr` methods, we will reach the recursive calls of the function at some point. We can then solve these by simple frame inference, if the correct pre- and postconditions were provided (Figure 5.25).

```
method hnr_solve_recursive_call methods frame_match_atom =
  rule hnr_frame[rotated], assumption, hnr_frame_inference frame_match_atom
```

Figure 5.25: Solve recursive call method

5.9 General Hnr Automation

Having now all the parts together, we can construct the `hnr` automation. First, we create an Eisbach method for applying the `hnr` rules we created in section 5.5 and a custom rule set, where we will provide our (diff) array rules (Figure 5.26). The custom rules are applied framed with the frame inference running afterward.

Next, we put the `hnr` rule, keep drop, normalize, merge, monotonicity solver, fallback,

```
method hnr_rule methods frame_match_atom uses rule_set =
  (rule rule_set[framed] hnr_pass[framed], hnr_frame_inference frame_match_atom)
  | rule hnr rule hnr const
```

Figure 5.26: Hnr rule method

and recursive call solver methods in this order together so that the first method that can be applied is applied (Figure 5.27).

Doing this as often as possible completes our translation automation (Figure 5.28).

```
method hnr_step methods frame_match_atom keep_atom uses rule_set normalization_rules =
     simp only: let_to_bind split_def
   | hnr_rule frame_match_atom rule_set: rule_set
   | keep_drop keep_atom
   | normalize rules: normalization_rules
   | merge rules: normalization_rules
   | partial_function_mono
   | hnr_fallback
   | hnr_solve_recursive_call frame_match_atom
```

Figure 5.27: Hnr step method

```
method hnr methods frame_match_atom keep_atom uses rule_set normalization_rules =
  (hnr_step frame_match_atom keep_atom
      rule_set: rule_set normalization_rules: normalization_rules)+
```

Figure 5.28: Hnr method

5.10 Hnr Array

Since we do not just want to transfer programs from the option monad to the heap monad but also use the features of the heap, we will create `hnr` rules for converting lists to arrays and diff arrays (section 5.11). To mark lists that should be converted to arrays, we create the definition in Figure 5.29.

If we reach such a definition, we create an array and introduce an array assertion using the rule in Figure 5.30.

We collect the array rules in a rule set called `hnr_rule_arr`, which we then pass on to the `hnr` automation methods. The other rules are similarly straightforward and follow the rule of having an assertion for every variable in the program. Moreover, they use safe versions of the standard array operation of Imperative/HOL, which we created analogous to the safe versions of the diff array operations (subsection 4.4.5), so we do

```
definition New_Arr :: "'a list ⇒ 'a list" where
  "New_Arr xs = xs"
```

Figure 5.29: Hnr array marker

```
lemma hnr_of_list [hnr_rule_arr]:
  "hnr
    emp
    (Array.of_list xs)
    array_assn
    (Some (New_Arr xs))"
```

Figure 5.30: Hnr array constructor

not have to deal with the bounds of the arrays.

```
lemma hnr_lookup [hnr_rule_arr]:
   "hnr
     (xsi ↦a xs * id_assn i ii)
     (Array_Safe.lookup xsi ii)
     (λr ri. array_assn xs xsi * id_assn r ri)
     (Some (xs ! i))"

lemma hnr_update [hnr_rule_arr]:
   "hnr
      (array_assn xs xsi * id_assn i ii * id_assn v vi)
      (Array_Safe.update ii vi xsi)
      array_assn
      (Some (xs [i:= v]))"

lemma hnr_length [hnr_rule_arr]:
   "hnr
      (array_assn xs xsi)
      (Array.len xsi)
      (λr ri. array_assn xs xsi * id_assn r ri)
      (Some (length xs))"

lemma hnr_pass_arr [hnr_rule_arr]:
  "hnr (array_assn x xi) (return xi) array_assn (Some x)"
```

Figure 5.31: Hnr array rules

All the rules can be directly proven by converting the `hnr` statements to Hoare triples and then using the separation logic proof automation. Moreover, by passing the rule set to the `hnr` proof automation (section 5.9) and using entailment reflexivity as strategies for the frame inference and keep drop method, we can finally convert programs using lists to arrays with verified equivalence.

As an example, we translate Figure 5.3, which results in Figure 5.33, where the list length operation and the list update operations are replaced with the corresponding array operations.

More examples are in the theory `Test_Hnr.thy` (Toth 2022).

```
method ent_refl = rule ent_refl

method hnr_arr = hnr ent_refl ent_refl rule_set: hnr_rule_arr
```

Figure 5.32: Hnr array method

```
synth_definition example_1_arr is [hnr_rule_arr]:
  "hnr (array_assn xs xsi) (¤ :: ?'a Heap) ?Γ' (example_1 xs)"
  unfolding example_1_def
  by hnr_arr
```

```
Output:
example_1_arr ≡
Array.len xsi >>=
(λx. return 2 >>=
   (λxa. return (x < xa) >>=
       (λx. if x
           then return 1 >>=
               (λx. Array_Safe.update x x xsi >>= return)
           else return 2 >>=
               (λx. Array_Safe.update x x xsi >>= return)
       )
    )
)
```

Figure 5.33: Example translation to arrays

The conversion works unless the list is used in a non-linear way since arrays are ephemeral. To also cover this case, we will use diff arrays.

5.11 Hnr Diff Array

The `hnr` rules for diff arrays are analogous to the array ones, except for the array assertion, instead of which we use a new version of the master assertion (Figure 5.34).

```
definition master_assn' where
  "master_assn' S = (∃A t. master_assn t * ↑(∀ (xs, xsi) ∈ S. t ⊢ xs ~ xsi))"
```

Figure 5.34: Hnr master assertion

This new version already contains the diff array relation and takes a set of diff arrays and its corresponding list abstractions. These represent all the available versions of the diff array. Further, we ensure again that each variable in each rule has an assertion and collect all the rules in a rule set (Figure 5.35). Also, we introduce a definition with the name `New_Diff_Arr` to mark places where a diff array should be created.

```
lemma hnr_pass_diff_arr [hnr_rule_diff_arr]:
  "hnr
      (master_assn' (insert (xs, xsi) S))
      (return xsi)
      (λxs' xsi'. master_assn' (insert (xs', xsi') S))
      (Some xs)"

definition New_Diff_Arr where
  "New_Diff_Arr a = a"

lemma hnr_from_array [hnr_rule_diff_arr]:
  "hnr
    (array_assn xs xsi)
    (Diff_Arr.from_array xsi)
    (λxs xsi. master_assn' { (xs, xsi) })
    (Some (New_Diff_Arr xs))"

lemma hnr_from_list [hnr_rule_diff_arr]:
  "hnr
    emp
    (Diff_Arr.from_list xs)
    (λxs xsi. master_assn' { (xs, xsi) })
    (Some (New_Diff_Arr xs))"

lemma hnr_lookup [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) S) * id_assn i ii)
    (Diff_Arr_Safe.lookup xsi ii)
    (λr ri. id_assn r ri * master_assn' S)
    (Some (xs ! i))"

lemma hnr_realize:
  "hnr
    (master_assn' (insert (xs, xsi) S))
    (Diff_Arr.realize xsi)
    (λ r ri. master_assn' S * array_assn r ri)
    (Some xs)"

lemma hnr_update [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) S) * id_assn i ii * id_assn v vi)
    (Diff_Arr_Safe.update xsi ii vi)
    (λxs' xsi'. master_assn' (insert (xs', xsi') S))
    (Some (xs [i:= v]))"

lemma hnr_length [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) S))
    (Diff_Arr.length xsi)
    (λr ri. master_assn' S * id_assn r ri)
    (Some (length xs))"
```

Figure 5.35: Hnr diff array rules

The proofs are again done by converting the `hnr` statements to Hoare triples and then using separation logic proof automation, which relies on the Hoare triple rules from section 4.4.

5.11.1 Set Inference

This time we cannot just use reflexivity as the strategy for matching assertions in the frame and for the keep drop method. Simple reflexivity cannot solve goals for inserting elements into the set of the master assertion because the insert operations can be in different orders. We build up a procedure for this case, which we call *set inference* and pass it as the matching strategy to our previously implemented frame inference. The set inference works on rules of the form: `master_assn' S` $\Longrightarrow_A$ `master_assn' S'`.
An example for `S` could be `S` $=$ { `a, b, c`} and for `S'`: `S'` $=$ `insert b ?F`. Then, we need to determine if all the elements we know in `S'`, and therefore do not belong to the frame, are in `S`. For that, we use a similar approach as for the frame inference previously. As the first step of the set inference, we notice that for solving a goal of the described form, it is enough to solve a goal of the form `S = S'` by applying the rule in Figure 5.36.

```
lemma master_assn'_cong:
  assumes
    "S = S'"
  shows
    "master_assn' S ⟹A master_assn' S'"
```

Figure 5.36: Master assertion congruence rules

Next, we introduce an additional empty set on the left of the equation to rotate the elements, similar to the frame inference. Then on the right-hand side, we introduce a tag as a trick to see whether all known elements are already matched. For this, we first use rule Figure 5.37 and then move the tag by applying rule Figure 5.38 multiple times to a position, such that all the known elements are outside of it.

```
definition Si_Tag where
  "Si_Tag x = x"

lemma si_initialize: "A ∪ {} = Si_Tag B ⟹ A = B"
```

Figure 5.37: Set inference initialization

```
lemma si_move_tag: "Si_Tag (insert x B) = insert x (Si_Tag B)"
```

Figure 5.38: Move set inference tag

```
lemma si_match: "insert x A ∪ B = C ⟹ insert x A ∪ B = insert x C"
```

Figure 5.39: Match elements in set inference

Applied to our example we would transform:
$\{a, b, c\} = \texttt{insert b ?F}$ to $\{a, b, c\} \cup \{\} = \texttt{insert b (Si\_Tag ?F)}$.
Now we take the first element on the left side and compare it to the outermost element on the right side. If they are the same, then we can match them. Otherwise, we put the element on the left into the second set and continue with the next element. We do the matching using the rule in Figure 5.39, and for the rotation, we use Figure 5.40.

```
lemma si_rotate: "A ∪ insert x B = C ⟹ insert x A ∪ B = C"
```

Figure 5.40: Rotate element in set inference

When we find a match, we rotate the left side back to its initial state (Figure 5.41). Then, we either stop when reaching the tag on the right side (Figure 5.42), meaning we have matched all assertions except the frame, or cancel the matching if we cannot match one of the elements.

```
lemma si_rotate_back: "insert x A ∪ B = C ⟹ A ∪ insert x B = C"
```

Figure 5.41: Rotate elements back for matching next element in set inference

```
lemma si_finish: "A ∪ {} = Si_Tag A"
```

Figure 5.42: Successful set inference

In the just-described way, we put these rules together using Eisbach methods (Figure 5.43).

```
method si_try_match = then_else
  ‹rule si_match›
  ‹((rule si_rotate_back)+)?›
  ‹rule si_rotate, si_try_match›

method si_initialize = rule si_initialize, (simp(no_asm) only: si_move_tag)?

method set_inference_keep = si_initialize, ((rule si_finish | si_try_match)+)?

method set_inference = set_inference_keep; fail

method hnr_diff_arr_match_atom = then_else
  ‹rule master_assn'_cong›
  ‹set_inference›
  ‹rule ent refl›
```

Figure 5.43: Set inference methods

5.11.2 Keep - Drop Strategy

In our keep drop procedure (section 5.2), we will reach goals for diff array assertions of the form: `master_assn' S` $\Longrightarrow_A$ `master_assn' ?S`. Here, we try to keep the assertion by keeping as many elements inside the master assertion as possible. So if we have, for example:

$\bigwedge$ `a b c. master_assn' ({ a, b, c }) ` $\Longrightarrow_A$ `master_assn' (?S a c)`.

Then the version `b` of the diff array goes out of scope, but `a` and `c` stay in scope since `?S` can depend on them. This could happen when `b` was, for example, defined inside an `if`-statement, and we leave its body. Like in subsection 5.11.1, reflexivity is not enough here to solve the entailment. Rather, we need to find the biggest (non-empty) subset of `S`, that we can get according to the possible dependencies of `?S`. We introduce such a subset relation using the rule in Figure 5.44, which also introduces a simple identity goal. This goal will ensure later that the subset we found is not empty.

```
lemma kdm_init:
  assumes
    "S' ⊆ S"
    "S' = S'"
  shows
    "master_assn' S ⟹A master_assn' S'"
```

Figure 5.44: Initialize keep-drop procedure for master assertions

To solve the subset relation, we go through all the elements in `S`, try to keep them (Figure 5.45), and if that is not possible, we drop them (Figure 5.46). For our example, this implies that we keep `a` and `c` but drop `b`.

```
lemma kdm_keep:
  assumes
    "S' ⊆ S"
  shows
    "insert x S' ⊆ insert x S"
```

Figure 5.45: Keep element of master assertion

```
lemma kdm_drop:
  assumes
    "S' ⊆ S"
  shows
    "S' ⊆ insert x S"
```

Figure 5.46: Drop element of master assertion

We put this strategy together in an Eisbach method (Figure 5.47), where we apply the keep and drop rules as often as possible and resolve the goal afterward using reflexivity. Then we check, using the previously introduced identity goal, that `S'` is not empty (Figure 5.48) because having an empty master assertion means that the two assertions should not have been matched in the first place. Finally, we put these steps again together using Eisbach methods (Figure 5.49).

```
method kdm_subset = ((rule kdm_keep | rule kdm_drop)+)?, rule subset_refl
```

Figure 5.47: Resolve subset relation

```
method kdm_check_not_empty = then_else
  ‹rule refl[of "{}"]›
  ‹fail›
  ‹rule refl›
```

Figure 5.48: Check that the found subset is not empty

```
method kdm = rule kdm_init, kdm_subset, kdm_check_not_empty

method diff_arr_kdm = rule ent_refl | kdm
```

Figure 5.49: Keep - drop procedure for master assertions

By passing this strategy, the strategy of subsection 5.11.1, the diff array `hnr` rules, and some normalization rules for the set of the master assertion (Figure 5.35) to the general `hnr` automation method (section 5.9), we complete the automatic refinement procedure of lists to diff arrays.

```
method hnr_diff_arr =
  hnr hnr_diff_arr_match_atom diff_arr_kdm
    rule_set: hnr_rule_diff_arr
    normalization_rules: insert_commute insert_absorb2
```

Figure 5.50: Hnr diff array method

6 Example: Lomuto Partitioning Algorithm

As an example application of our refinement procedure, we implement the Lomuto partitioning algorithm naively (Bentley 1984, p.287), except that instead of having the pivot element at the end, we put it into the first place of the list (Figure 6.1). The implementation is naive, because it simply replaces the arrays in the imperative implementation with functional lists, using the same operations. Due to that, the runtime of the algorithm increases from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.

```
definition swap :: "nat ⇒ nat ⇒ 'a list ⇒ 'a list" where
  "swap i j xs ≡ (xs[i := xs!j])[j := xs!i]"

fun partition :: "nat ⇒ nat ⇒ ('a::linorder) list ⇒ ('a list * nat)" where
  "partition i j xs = (if 1 < j then
      (if xs ! 0 < xs ! (j - 1)
       then partition (i - 1) (j - 1) (swap (i - 1) (j - 1) xs)
       else partition i (j - 1) xs)
    else (swap (i - 1) 0 xs, i - 1)
  )"

abbreviation partition' where
  "partition' xs ≡ partition (length xs) (length xs) xs"
```

Figure 6.1: Naive Lomuto partitioning algorithm

By recursing on the list and returning a separate list per partition, like in the Isabelle standard library (Nipkow 2022), we can also reach a runtime in $\mathcal{O}(n)$. But it is not possible to reach the same runtime for every algorithm compared to its imperative counterpart when implemented purely functionally, since some have a logarithmic slowdown (Pippenger 1997, p.108).

With our refinement procedure, we do not need to rewrite the algorithm and still regain a runtime in $\mathcal{O}(n)$ for the Lomuto partitioning. As a first step for that, we verified the implementation by finding the invariant and doing a structural induction (Figure 6.2). Next, we need to monadify the implementation (see chapter 5). For now, we do this

```
definition inv :: "nat ⇒ nat ⇒ ('a::linorder) list ⇒ bool" where
  "inv i j xs ≡
    let p = xs ! 0 in
    0 < length xs ∧
    0 < i ∧
    i ≤ length xs ∧
    j ≤ length xs ∧
    j ≤ i ∧
   (∀h ∈ set (drop i xs). p < h) ∧
   (∀l ∈ set (take (i - j) (drop j xs)). l ≤ p)"

definition is_valid_partition where
  "is_valid_partition ys m ≡ ∀l ∈ set (take m ys). ∀h ∈ set (drop m ys). l ≤ h"

lemma partition_correct_elements: "partition i j xs = (ys, m) ⟹ set xs = set ys"

lemma partition: "inv i j xs ⟹ partition i j xs = (ys, m) ⟹ is_valid_partition ys m"

lemma partition':
  "partition' (p#xs) = (ys, m) ⟹ is_valid_partition ys m"
```

Figure 6.2: Verification of the naive Lomuto partitioning algorithm

manually (chapter 7). Additionally, we verified that the monadified algorithm does the same as the original one. The monadified version of `swap` is in Figure 6.3 and the whole monadified algorithm is in Toth 2022.

```
definition swap_opt :: "nat ⇒ nat ⇒ 'a list ⇒ 'a list option" where
  "swap_opt i j xs = do {
     let c1 = xs!j;
     let c2 = xs!i;
     let c3 = xs[i := c1];
     let c4 = c3[j := c2];
     Some c4
  }"

lemma swap_opt_termination: "swap_opt i j xs = Some (swap i j xs)"
```

Figure 6.3: Monadified swap implementation

Now, we can start with the actual refinement process. The algorithm can be refined using arrays and diff arrays since the utilized list is used linearly. But if we change, for example, the order of the bindings of `c2` and `c3` in Figure 6.3, the list would not be used linearly anymore and we would need a diff array.

The implementation of `swap` can be refined directly using the `hnr_diff_arr` method (Figure 5.50). For `partition`, we first need to apply the recursion method (section 5.8) and supply the pre- and postcondition of the recursive calls in form of separation logic assertions (Figure 6.4). They are straightforward and state that the parameters of the function as well as the result of the recursive calls are either identical to their refinements

or in the case of the list it is refined to the respective diff array. Finally, our diff array refinement method (Figure 5.50) can do the rest of the refinement automatically and replaces the list with diff arrays and its corresponding operations.

```
synth_definition swap_impl is [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) F) * id_assn i ii * id_assn j ji)
    (⌑:: ?'a Heap)
    ?Γ'
    (swap_opt i j xs)"
  unfolding swap_opt_def
  by hnr_diff_arr

synth_definition partition_impl is [hnr_rule_diff_arr]:
  "hnr
    (master_assn' (insert (xs, xsi) F) * id_assn i ii * id_assn j ji)
    (⌑:: ?'a Heap)
    ?Γ'
    (partition_opt (i, j, xs))"
  unfolding partition_opt_def
  apply(hnr_recursion
          "(λF p pi.
                master_assn' (insert (snd(snd p), snd (snd pi)) F) *
                id_assn (fst p) (fst pi) *
                id_assn (fst (snd p)) (fst (snd pi)))"
          "(λF p pi r ri.
                master_assn' (insert (snd(snd p), snd (snd pi))
                             (insert (fst r, fst ri) F)) *
                id_assn (snd r) (snd ri) *
                id_assn (fst p) (fst pi) *
                id_assn (fst (snd p)) (fst (snd pi)) *
                true
                )"
          hnr_diff_arr_match_atom
       )
  by hnr_diff_arr
```

Figure 6.4: Refinement of the naive Lomuto partitioning

7 Future Work

We need to address the following points to have a fully automated and verified refinement process of functional lists to (diff) arrays:

1. The input programs are not yet automatically monadified into the format described in chapter 5. However, we could do this similar to the approach of Wimmer, Hu, and Nipkow 2018, p.2.

2. Currently, we need to manually provide the pre- and postconditions for translations of recursive calls. As a possible solution outline, we could require that the program's terminating branches occur before the recursive ones and generate out of their known assertions the missing pre- and postconditions.

3. We cannot yet refine the creation of types containing already refined types. For example, to translate a list of lists to a diff array of diff arrays, we would need new assertions types, which consider the nesting. We would, inter alia, need to relate `cells` of `cells` with `cell`'s of `cell`'s similar to Figure 4.3 but also accounting for the nesting. We have yet to discover if and how this could be generalized to arbitrarily nested types.

4. The translation of `case`-statements are not yet generalized, so every newly defined algebraic data type needs its own hnr rule following the schema of subsection 5.5.4. A generalization will need a deeper dive into the inductive datatypes of Isabelle.

Another interesting extension of the current framework would be a linearity analysis of the input programs, which determines if lists that our framework will refine are used linearly or not. Like that, we could decide if a list should be refined to a diff array or if

we can avoid its (small) overhead and refine it to a plain array. A similar approach to Bernardy, Boespflug, Newton, et al. 2017 would be conceivable.
Also, a generalization of the framework as a facility for many kinds of refinements is imaginable.

List of Figures

Bibliography

Bentley, J. (1984). "Programming pearls: how to sort." In: *Communications of the ACM* 27.4, 287–ff.

Bernardy, J.-P., M. Boespflug, R. R. Newton, S. P. Jones, and A. Spiwack (2017). "Linear Haskell: practical linearity in a higher-order polymorphic language." In: DOI: 10.48550/ARXIV.1710.09756.

Bloss, A. (1989). "Update analysis and the efficient implementation of functional aggregates." In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*. ACM Press. DOI: 10.1145/99370.99373.

Bulwahn, L., A. Krauss, F. Haftmann, L. Erkök, and J. Matthews (2008). "Imperative Functional Programming with Isabelle/HOL." In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 134–149. DOI: 10.1007/978-3-540-71067-7_14.

Calcagno, C., P. W. O'Hearn, and H. Yang (2007). "Local Action and Abstract Separation Logic." In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE. DOI: 10.1109/lics.2007.30.

Krauss, A. (2010). "Recursive Definitions of Monadic Functions." In: DOI: 10.48550/ARXIV.1012.4895.

Kumar, A., G. E. Blelloch, and R. Harper (May 2017). "Parallel functional arrays." In: *ACM SIGPLAN Notices* 52.1, pp. 706–718. DOI: 10.1145/3093333.3009869.

Lammich, P. (Oct. 2017). "Refinement to Imperative HOL." In: *Journal of Automated Reasoning* 62.4, pp. 481–503. DOI: 10.1007/s10817-017-9437-1.

– (2019). "Generating Verified LLVM from Isabelle/HOL." en. In: DOI: 10.4230/LIPICS.ITP.2019.22.

Lammich, P. and R. Meis (Nov. 2012). "A Separation Logic Framework for Imperative HOL." In: *Archive of Formal Proofs*. https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development. ISSN: 2150-914x.

Launchbury, J. and S. L. P. Jones (Dec. 1995). "State in Haskell." In: *LISP and Symbolic Computation* 8.4, pp. 293–341. DOI: 10.1007/bf01018827.

Nipkow, T. (2013). "Programming and proving in Isabelle/HOL." In: *Technical report, University of Cambridge*.

– (2022). *List - Isabelle Standard Library*. https://github.com/seL4/isabelle/blob/master/src/HOL/List.thy.

Okasaki, C. (Apr. 1998). *Purely Functional Data Structures*. Cambridge University Press. DOI: 10.1017/cbo9780511530104.

Pippenger, N. (Mar. 1997). "Pure versus impure Lisp." In: *ACM Transactions on Programming Languages and Systems* 19.2, pp. 223–238. DOI: 10.1145/244795.244798.

Reynolds, J. (2002). "Separation logic: a logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. DOI: 10.1109/lics.2002.1029817.

Toth, B. (2022). *Arrays in Isabelle - Repository*. https://github.com/balazstothofficial/arrays-in-isabelle.

Wenzel, M. et al. (2004). *The isabelle/isar reference manual*.

Wimmer, S., S. Hu, and T. Nipkow (2018). "Verified Memoization and Dynamic Programming." In: *Interactive Theorem Proving*. Springer International Publishing, pp. 579–596. DOI: 10.1007/978-3-319-94821-8_34.