


Refinement with Time – Refining the Run-time of Algorithms in Isabelle/HOL

Maximilian P. L. Haslbeck 

Technisch Universität München, Germany

<https://www21.in.tum.de/~haslbema/>

haslbema@in.tum.de

Peter Lammich 

The University of Manchester, England

peter.lammich@manchester.ac.uk

Abstract

Separation Logic with Time Credits is a well established method to formally verify the correctness and run-time of algorithms, which has been applied to various medium-sized use-cases. Refinement is a technique in program verification that makes software projects of larger scale manageable.

Combining these two techniques, we present a methodology for verifying the functional correctness and the run-time analysis of algorithms in a modular way. As use-cases we verify Kruskal’s minimum spanning tree algorithm and the Edmonds–Karp algorithm for Network Flow.

An adaptation of the Isabelle Refinement Framework [15] enables us to specify the functional result and the run-time behaviour of abstract algorithms which can be refined to more concrete algorithms. From these, executable imperative code can be synthesized by an extension of the Sepref tool [12], preserving correctness and the run-time bounds of the abstract algorithm.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Separation logic; Theory of computation → Logic and verification

Keywords and phrases Isabelle, Time Complexity Analysis, Separation Logic, Program Verification, Refinement, Run Time, Algorithms

Digital Object Identifier 10.4230/LIPIcs.XYZ.2019.23

Funding Maximilian P. L. Haslbeck: DFG Grant NI 491/16-1

Peter Lammich: DFG Grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases"

Acknowledgements We want to thank Simon Wimmer and Armaël Guéneau for useful suggestions to improve the paper.

1 Introduction

Recently the literature has seen various interactive verification efforts for run-time analysis of efficient algorithms and data structures: Charguéraud et al. [4] verify the union-find data structure, Zhan et al. [17] formalize amongst others the median of medians selection algorithm, Karatsuba’s algorithm and splay trees, and most recently Guéneau et al. [8] verify a state-of-the-art incremental cycle detection algorithm.

While the largest of these developments fits on one page (Figure 1 in [8]) more ambitious projects have been tackled when only functional correctness is concerned: Esparza et al. [5] formalized a LTL-model checker, Fleury et al. [6] verified a SAT-solver, Wimmer et al. [16] formalized a timed automaton model checker, various graph algorithms have been verified [11, 13]. The list is growing. One key ingredient to manage the complexity of larger algorithm developments is to use refinement. It allows to separate reasoning about the abstract algorithmic idea from reasoning about implementation details. In the Isabelle world, the Isabelle Refinement Framework [15] can be used to express abstract algorithms and to



© Maximilian P. L. Haslbeck and Peter Lammich;
licensed under Creative Commons License CC-BY

XYZ 2019.

Editors: -, Article No. 23; pp. 23:1–23:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

45 use step-wise refinement to form concrete algorithms. As a last step the Sepref tool [12] can
 46 be used as a back end to synthesize efficient executable imperative code while preserving
 47 correctness. Target languages for that tool are hybrid languages such as SML, Scala and
 48 more recently the purely imperative language LLVM [10]. From such verification efforts one
 49 can obtain executable algorithms that are often competitive with real world implementations
 50 within one order of magnitude. However, only functional correctness is ensured and not
 51 run-time bounds.

52 This paper brings together run-time analysis and refinement. By extending the refinement
 53 approach to also reason about the run-time of algorithms in a modular and scalable way, we
 54 lay the ground for an additional run-time analysis of larger algorithms.

55 Our vision is to specify abstract algorithms and their run-time in terms of abstract
 56 operations with time bounds — say Edmonds–Karp algorithm uses at most $E * V$ find-
 57 augmenting-path operations. When we then refine an operation like find-augmenting-path to
 58 a more concrete BFS algorithm involving operations such as set membership test and map
 59 lookup, we can also refine the abstract compound algorithm to use the more refined operation.
 60 Just as for plain refinement we separate abstract run-time arguments from reasoning about
 61 run-times of concrete data structures. As a last step we synthesize executable imperative
 62 code which refines the abstract algorithm and thus obeys both the high-level correctness
 63 theorem and the run-time bound.

64 The main contributions of this paper are:

- 65 • We build a theory for refinement with time by creating NREST, the non-determinism
 66 monad with time, together with tools for reasoning about programs in that monad
 67 (Section 2).
- 68 • We extend the Sepref Tool (Section 3.2) to synthesize executable imperative code in
 69 Imperative/HOL with time (Section 3.1) supporting imperative and amortized data
 70 structures seamlessly.
- 71 • We enable modular development of algorithms by providing a library of efficient amortized
 72 data structures and reusable algorithms with run-time guarantees (Section 4).
- 73 • We show the applicability of our approach to larger algorithm developments by use-cases
 74 such as Edmonds–Karp and Kruskal’s algorithm (Section 5).

75 The formalization described in this paper is available at <https://www21.in.tum.de/~haslbema/Sepreftime>.
 76

77 **2 Non-determinism Monad With Time**

78 In this section we introduce NREST, the timed non-determinism monad. It allows specifying
 79 the result and time consumption of programs. As this is an extension of the NRES monad of
 80 the Isabelle Refinement Framework, we follow Lammich [12] in some of our explanations.

81 **2.1 Timed Non-determinism Monad**

82 We want to specify the result of a computation together with its worst case execution time.
 83 We allow multiple results with one run-time bound each. Allowing non-determinism is a
 84 common technique in program refinement, used to hide implementation details of abstract
 85 algorithms. A program in the timed non-determinism monad is defined over the type
 86 α *NREST*:

87 $'a$ *NREST* = *RES* ($'a \Rightarrow$ *enat option*) | *FAIL*,

88 where *enat* is the type of extended natural numbers, i.e. $\mathbb{N} \cup \{\infty\}$ and $'a \Rightarrow 'b$ *option* is the
 89 type of a map from $'a$ to $'b$. The type α *NREST* describes non-deterministic results with
 90 time bounds, where *RES* M describes the non-deterministic choice of an element from the
 91 domain of M while consuming no more time units than M specifies for that element. *FAIL*
 92 describes a failed computation, which usually stems from an assertion that was not satisfied.

93 We define a *refinement ordering* on *NREST* by first lifting the ordering on *enat* to option
 94 with *None* as the bottom element, then pointwise to functions and finally to α *NREST*,
 95 setting *FAIL* as the top element. With that ordering, *NREST* forms a complete lattice where
 96 *RES* $(\lambda s. \text{None})$ is the bottom element, and *FAIL* is the top element. Intuitively, $N \leq M$
 97 means that program N *refines* program M , i.e. all results of N are also results of M , and
 98 further for each such result, N takes no more time than M does. Note that *FAIL* is refined
 99 by any program.

100 ► **Example 1.** A program that reverses a list and whose run-time is at most four times the
 101 length of the list can be specified by: *rev_spec* $xs = \text{RES } [\text{rev } xs \mapsto 4*|xs|]$

102 Here, $[a \mapsto b]$ is syntactic sugar for $(\lambda x. \text{if } x=a \text{ then } \text{Some } b \text{ else } \text{None})$.

103 On the type *NREST* we define the following functions:

```
104 consume :: 'a NREST  $\Rightarrow$  enat  $\Rightarrow$  'a NREST where
105 consume (RES M) t = RES ( $\lambda x. \text{case } M x \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } t' \Rightarrow \text{Some } (t + t')$ )
106 consume FAIL t = FAIL
```

```
109 return :: 'a  $\Rightarrow$  'a NREST where
110 return x = RES [ x  $\mapsto$  0 ]
```

```
113 bind :: 'a NREST  $\Rightarrow$  ('a  $\Rightarrow$  'b NREST)  $\Rightarrow$  'b NREST where
114 bind (RES M) f = Sup { consume (f x) t | x t. M x = Some t }
115 bind FAIL f = FAIL
```

118 The term *consume* $M t$ describes the computation M prolonged by t time steps, *return* x
 119 is a computation that yields a single result x in no time, and *bind* $m f$ is the sequential
 120 composition of two computations: First compute any result x of m , then any result y of $f x$.
 121 The time bounds for the final results have to be determined considering all possible ways how
 122 to reach them. If m or any reachable computation path of f fails the compound computation
 123 also fails.

124 *NREST* together with *bind* and *return* forms a monad and *bind* as well as *consume* are
 125 monotonic w.r.t. the refinement ordering:

```
126 m  $\leq$  m'  $\longrightarrow$  ( $\forall x. f x \leq f' x$ )  $\longrightarrow$  bind m f  $\leq$  bind m' f
127 m  $\leq$  m'  $\longrightarrow$  t  $\leq$  t'  $\longrightarrow$  consume m t  $\leq$  consume m' t'
```

130 ► **Example 2.** Let $m = \text{RES } (\lambda_::\text{nat. } \text{Some } 0)$ and $f v = \text{consume } (\text{return } 0) v$. Program
 131 m computes any natural number in no time, and f takes a natural number v as argument and
 132 computes the result 0 in at most v steps. Now consider *bind* $m f$. Both m and f do not fail,
 133 and together compute the single result 0 . But there are computation paths (via any value v
 134 produced by m) with any natural number as a run-time. The supremum over all these is
 135 ∞ . To sum it all up: *bind* $m f = \text{consume } (\text{return } 0) \infty$. This illustrates why we had to
 136 choose *enat* for the range of the run-time bound, rather than the type of natural numbers.

137 Furthermore we define two derived operations:

```

138
139 SPEC :: ('a ⇒ bool) ⇒ ('a ⇒ enat) ⇒ 'a NREST where
140 SPEC P t = RES (λv. if P v then Some (t v) else None)
141

```

```

142
143 assert :: bool ⇒ unit NREST where
144 assert P = (if P then return () else FAIL)
145

```

146 A computation that returns a result v if and only if $P v$ holds and takes at most $t v$ time
147 is described by $SPEC P t$. The computation `assert P` fails if the predicate P is not satisfied.
148 For assertions we have the following rules:

```

149
150 P → m ≤ m' → do { assert P; m } ≤ m'
151 (P → m ≤ m') → m ≤ do { assert P; m' }
152

```

153 Here, we use a Haskell-like `do` notation as a convenient syntax for bind operations. The
154 first rule is used to show that a program m with assertion P refines the program m' . It
155 requires to prove P , in addition to the refinement $m \leq m'$. The second rule is used to show
156 that a program m refines a program m' with an assertion. It allows one to assume P when
157 proving the refinement $m \leq m'$. This way, facts that are proven on the abstract level are
158 made available for proving refinement.

159 2.2 Recursive Programs

160 Non-recursive programs can be expressed by the monad operations and Isabelle/HOL's if
161 and case-combinators. Recursion is encoded by a fixed point combinator $RECT$, such that
162 $RECT F$ is the greatest fixed point of the monotonic functor F , w.r.t. to the flat ordering of
163 timed result maps with $FAIL$ as the top element. For any non-monotonic F , $RECT F$ is set
164 to $FAIL$:

```

165
166 RECT :: (('b ⇒ 'a NREST) ⇒ 'b ⇒ 'a NREST) ⇒ 'b ⇒ 'a NREST where
167 RECT F x = (if mono2 F then (gfp F x) else FAIL)
168

```

169 Here, $mono2$ denotes monotonicity w.r.t. both the flat ordering and the refinement
170 ordering. The benefits of this are explained in more detail elsewhere [12]. Note that
171 programs constructed by the combinators we introduced above are monotonic in that sense
172 by construction. The combinator $RECT$ is also monotonic w.r.t. the refinement ordering:

```

173
174 mono2 B ∧ (∀F x. B F x ≤ B' F x) → RECT B x ≤ RECT B' x
175

```

176 For all other combinators we can show similar monotonicity lemmas. Building on them,
177 we also define while loops, foreach loops and a fold function to conveniently express tail
178 recursion, folding over the elements of a finite set and folding over a list.

179 ► **Example 3.** As a running example we consider the formalization of Kruskal's algorithm.
180 To illustrate the expressive power of NREST we present the abstract algorithm in Figure
181 1a: the greedy algorithm to construct a minimum weight basis for a matroid. This abstract
182 algorithm will later be instantiated for the cycle matroid, which yields the skeleton of
183 Kruskal's algorithm. Already on this abstract level we can structure the algorithm and prove
184 the functional correctness of the algorithmic idea, as well as its run-time — parameterized
185 over the run-times of the abstract operations it performs.

186 In line two the algorithm obtains a list of the elements of the carrier set E (later this will
187 be the set of edges of an undirected graph) sorted w.r.t. some weight function w . Starting from

<pre> 1 minWeightBasis = do { 2 l ← SPEC (λL. sorted_wrt w L 3 ∧ distinct L ∧ set L = E) 4 (λ_. t_{sc}); 5 s ← RES [∅ ↦ t_{eb}]; 6 T ← nfold l (λe T. do { 7 assert (e ∉ T ∧ indep T ∧ e ∈ c ∧ T ⊆ E); 8 b ← RES [indep (T ∪ {e}) ↦ t_{it}]; 9 if b then do { 10 11 RES [T ∪ {e} ↦ t_i] 12 } else 13 return T 14 }) s; 15 return T 16 }</pre>	<pre> 1 Kruskal = do { 2 l ← obtain_sorted_edge_list; 3 4 5 (djs0, fl0) ← initState; 6 (djs, fl) ← nfold l (λ(a,w,b) (djs, fl). do { 7 assert (a ∈ Domain djs ∧ b ∈ Domain uf); 8 b ← RES [¬djs_cmp djs a b ↦ t_{it}]; 9 if b then do { 10 assert ((a,w,b) ∉ set fl); 11 addEdge djs a b fl 12 } else 13 return (djs,fl) 14 }) (djs0, fl0); 15 return fl 16 }</pre>
--	---

(a) The greedy algorithm to construct a minimum weight basis of a Matroid in the NREST monad. (b) A further refinement for the Kruskal algorithm, where an additional disjoint sets datastructure is passed around.

■ Figure 1

188 an empty independent set, we iteratively add elements if they leave the set T independent
189 (i.e. create no cycle in the graph case). For all operations that may cost time, we reserve
190 some time parameter of type *nat* or functions to *nat*: here t_{sc} , t_{eb} , t_{it} and t_i stand for sorted
191 carrier set time, empty basis time, independence test time and insertion time.

192 We can give the specification for this algorithm, and state the refinement theorem:

193
$$\text{minWeightBasis} \leq \text{SPEC minBasis} (\lambda_. t_{sc} + t_{sb} + |E| * (t_{it} + t_i))$$

194
195

196 where *minBasis* S is true iff S is a minimum weight basis. How to prove such a refinement in
197 a mechanized way is the subject of the next section.

198 2.3 Generalizing the Weakest Precondition

199 First let us consider refinement goals with a result on the right hand side: $c \leq \text{RES } Q$

200 That is, we want to prove that a program c meets specification Q . Note that program c
201 might be a composed program using the combinators defined above. In order to come up
202 with meaningful rules for these combinators we first need to generalize the above goal.

203 Instead of asking only *whether* a program satisfies the specification, we also ask “*how*
204 *much*” it satisfies the specification, i.e. how much slack time is between the specified and
205 actual run-time. As a mental model, we place the “slack time” *in front* of the actual run-time
206 and call it the *latest starting time* such that executing c always terminates before the deadline
207 $Q :: 'a \Rightarrow \text{enat option}$, and denote it as $\text{lst } c \ Q :: \text{enat option}$.

208 If program c does not fulfill a specification Q then there is no such time and $\text{lst } c \ Q = \text{None}$,
209 otherwise its value is the latest feasible starting time. Before we give the definition of *lst*, let
210 us explore what we can do with it. We obtain the following equality:

211
$$c \leq \text{RES } Q \iff \text{Some } 0 \leq \text{lst } c \ Q$$

212
213

23:6 Refinement with Time

214 and we can prove the following equation for the bind operator:

$$215 \quad \text{lst} (\text{bind } m \ f) \ Q = \text{lst } m \ (\lambda y. \text{lst } (f \ y) \ Q)$$

218 Intuitively it says: The latest starting time for the compound computation `bind m f` to
219 satisfy Q is the latest starting time for m in order to meet the latest starting time such that
220 $f \ y$ meets the specification Q .

221 To determine $\text{lst } c \ Q$, we need to consider the differences between the specified and the
222 actual run-time for every result of c and take the most conservative one:

$$223 \quad \text{lst } c \ Q = \text{Inf } r. \text{minus } Q \ c \ r$$

226 Operation $\text{minus} :: ('a \Rightarrow \text{enat option}) \Rightarrow 'a \ \text{NREST} \Rightarrow 'a \Rightarrow \text{enat option}$ formalizes tak-
227 ing the difference. We have the following cases:

- 228 – $c = \text{RES } C$ and $C \ r = \text{None}$: as C will never produce the result r , it can be ignored, i.e.
229 $\text{minus } Q \ c \ r = \text{None}$.
- 230 – $c = \text{RES } C$ and $C \ r = \text{Some } m$ and $Q \ r = \text{None}$: r is specified to not be obtained, but when
231 starting c we obtain r , thus there is no valid starting time for C : $\text{minus } Q \ c \ r = \text{None}$.
- 232 – $c = \text{RES } C$ and $C \ r = \text{Some } m$ and $Q \ r = \text{Some } n$: if more time is needed than specified
233 ($n < m$) there is no valid latest starting time and we return None , otherwise the difference
234 is returned ($\text{Some } (n - m)$).
235
236

237 We can get some more intuition when unfolding lst in the above equality:

$$238 \quad c \leq \text{RES } Q$$

$$239 \quad \longleftrightarrow \quad \text{Some } 0 \leq \text{lst } c \ Q \quad (= \text{Inf } r. \text{minus } Q \ c \ r)$$

$$240 \quad \longleftrightarrow \quad \forall r. \text{Some } 0 \leq \text{minus } Q \ c \ r$$

242 The infimum is just a compact version of saying that the difference of Q and c on *any*
243 result r is non-negative. By abusing notation and following the intuition of minus one can
244 restate the last line as “ $\forall r. c \ r \leq Q \ r$ ”. In essence it says, that c meets specification Q , iff
245 for any r the time that it takes to calculate r for c is at most the time that Q reserved for
246 that result.

247 2.4 The Refinement Rules are Sound

248 Instead of solving problems of the form $c \leq \text{RES } Q$ we solve problems of the more general
249 form $\text{Some } t \leq \text{lst } c \ Q$. This general form allows us to state syntax directed rules in a
250 uniform way, which would not be possible otherwise.

251 From the equality for lst on `bind` we can derive an introduction rule for `bind`:

$$252 \quad \text{Some } t \leq \text{lst } M \ (\lambda y. \text{lst } (f \ y) \ Q) \longrightarrow \text{Some } t \leq \text{lst } (\text{bind } M \ f) \ Q$$

255 For the other combinators we have:

$$256 \quad (\forall r \in M. \text{Some } (t + M \ r) \leq Q \ r) \longrightarrow \text{Some } t \leq \text{lst } (\text{RES } M) \ Q$$

$$257 \quad \text{Some } t \leq Q \ x \longrightarrow \text{Some } t \leq \text{lst } (\text{return } x) \ Q$$

$$258 \quad (\forall x. P \ x \longrightarrow \text{Some } (t + t' \ x) \leq Q \ x) \longrightarrow \text{Some } t \leq \text{lst } (\text{SPEC } P \ t) \ Q$$

$$259 \quad \text{Some } (t + t') \leq \text{lst } M \ Q \longrightarrow \text{Some } t \leq \text{lst } (\text{consume } M \ t') \ Q$$

262 For the fold operation `ifold` we have the following rule:

```

263  $I \sqcap l_0 s_0$ 
264  $\wedge (\forall x l_1 l_2 s. l_0 = l_1 \cdot [x] \cdot l_2 \wedge I l_1 ([x] \cdot l_2) s$ 
265  $\quad \longrightarrow \text{Some } 0 \leq \text{lst } (f x s) (\text{emb } (I (l_1 \cdot [x]) l_2) t_{\text{body}}))$ 
266  $\wedge (\forall s. I l_0 \sqcap s \longrightarrow \text{Some } (t + t_{\text{body}} * |l_0|) \leq Q s)$ 
267  $\longrightarrow \text{Some } t \leq \text{lst } (\text{nfold } l_0 f s_0) Q$ 
268
269
```

Here, $\text{emb } P t = (\lambda x. \text{if } P x \text{ then } \text{Some } t \text{ else } \text{None})$, nfold is defined in a straightforward manner and the invariant I is a predicate that takes as its first argument the list of already processed elements, then the list of elements still to be processed and finally a state s . For showing that $\text{nfold } l_0 f s_0$ meets its specification Q with slack time t , one has to show that an invariant I holds initially, the body preserves the invariant and takes at most t_{body} time steps and the invariant in the end implies the desired specification. As we fold over a finite list, a termination argument is not required.

We also define a rule for *RECT* and based on that one for while loops. With the above rules and analogous rules for *assert* and the combinators *if* and *case*, a syntax directed verification condition generator can easily be constructed by exhaustively applying those rules.

► **Example 4.** After annotating the loop in the abstract program from Figure 1b with $\text{body}_{\text{time}} = t_{\text{it}} + t_i$ and a suitable invariant $I = \lambda l_1 l_2 T. I_{\text{mwb}}(T, \text{set } l_2)$ (where $I_{\text{mwb}}(T, E)$ implies $\text{minBasis } T$ for the whole carrier set E), we run the VCG on the refinement theorem of Example 3 and obtain eleven verification conditions. One of these is the invariant preservation of the first branch of the if-expression, i.e. when adding an element e :

```

285  $\text{sorted\_wrt } w l \wedge \text{distinct } l \wedge \text{set } l = E \wedge l = l_1 \cdot [e] \cdot l_2 \wedge \text{indep } (T \cup \{e\})$ 
286  $\wedge I_{\text{mwb}}(T, \text{set } ([e] \cdot l_2)) \longrightarrow I_{\text{mwb}}(T \cup \{e\}, \text{set } l_2)$ 
287
288
289
```

This verification condition is one of the central ones in the correctness proof and can be discharged with an interactive proof.

2.5 Data Refinement

In the process of refining an abstract algorithm to a more concrete one, a usual task is to replace abstract data structures by concrete ones, for example to replace sets by lists. Consider the then branch in the algorithm in Figure 1a: instead of using a set to collect the elements of a basis, we want to represent this set by a list. We have the following refinement in mind. Given that a list l represents a set T (denoted by $(l, T) \in \text{list_set_rel}$), the resulting lists of the program on the left hand side refine the resulting sets produced by the right hand side program:

```

300  $(l, T) \in \text{list\_set\_rel} \longrightarrow \text{RES } [l \cdot [x] \mapsto \text{it}] \leq \Downarrow (\text{list\_set\_rel}) \text{RES } [T \cup \{x\} \mapsto \text{it}]$ 
301
302
```

Given a (single-valued) refinement relation R , i.e. a relation that maps a concrete element to some (a single) abstract elements, the concretization function $\Downarrow R$ maps abstract results to concrete results w.r.t. R .

```

306  $\Downarrow R \text{ FAIL} = \text{FAIL}$ 
307  $\Downarrow R (\text{RES } X) = \text{RES } (\lambda c. \text{Sup } \{X a \mid a. (c, a) \in R\})$ 
308
309
```

Data refinement is orthogonal to introducing the time counting, as it only acts on the domain of the maps, not on their values. We can lift all monotonicity lemmas to also include the data refinement, e.g. for the bind operation we obtain the following rule:

313 $M \leq \Downarrow R' M' \wedge (\forall x x'. (x, x') \in R' \longrightarrow f x \leq \Downarrow R (f' x')) \longrightarrow \text{bind } M f \leq \Downarrow R (\text{bind } M' f)$
 314

316 Analogous rules can be proven for *RECT*, *while*, *nfold*, *assert*, and the other combinators
 317 (like *if* and *case*).

318 2.6 Setting Up a VCG for Refinement

319 In practice, one mostly is confronted with two kinds of refinement goals: first, goals w.r.t.
 320 a specification $c \leq RES Q$, which we already considered, and second, refinement of two
 321 abstract algorithms that are structurally similar (c.f. Figure 1). For the latter case, one
 322 simulates the two programs in lock step and uses the monotonicity lemmas mentioned in
 323 the last section to divide and conquer the problem. Collecting these rules we construct an
 324 automated refinement solver, which we illustrate with an example:

325 ► **Example 5.** Consider the two programs in Figure 1. The concrete program *Kruskal* is
 326 a specialized minimum weight basis algorithm for the cycle matroid, where the elements
 327 of the matroid are edges in an undirected graph, represented by a tuple (a, w, b) of its end
 328 nodes a and b and weight w . Programs *obtain_sorted_edge_list* and *addEdge* are compound
 329 programs. We want to show the following refinement relation:

330 $Kruskal \leq \Downarrow \text{list_graph_rel } \text{minWeightBasis}$
 331
 332

333 where *list_graph_rel* relates a set of abstract edges in the graph with a list of edge tuples
 334 representing them. In the process of showing this refinement, several other intermediate
 335 refinement relations are used, e.g. $((djs, fl), T) \in djs_graph_rel$ which relates the abstract
 336 edge set T to the list of edges fl and its corresponding disjoint-sets data structure. The main
 337 part of this refinement proof is to show that testing independence if we add an edge (a, w, b)
 338 (i.e. checking cycle-freedom) can be implemented by comparing the equivalence classes of a
 339 and b .

340 Note that *addEdge* has to do two things: update the disjoint-sets data structure and add
 341 the edge tuple to the list. We specify this program abstractly, and reserve time t_{iu} and t_{il}
 342 for the two actions. In the refinement proof we need to prove that $t_{iu} + t_{il} \leq t_i$. Similarly,
 343 the sum of the costs in *obtain_sorted_edge_list* must be smaller than t_{sc} .

344 The VCG for refinement simulates the two programs side by side, using the monotonicity
 345 lemmas to split the problem into smaller parts, and then showing the refinements of those smaller
 346 parts. One such part is the goal $\text{addEdge } djs \ a \ b \ fl \leq \Downarrow \text{list_graph_rel } (RES [T \cup \{e\} \mapsto t_i])$
 347 (with *list_graph_rel* motivated as above).

348 3 Refinement to Imperative/HOL with Time

349 In this section we introduce the time-aware monad of Imperative/HOL [17], which we then
 350 use as the target monad of the adapted Sepref tool [12] with NREST as the source monad.

351 3.1 Imperative/HOL with Time

352 Imperative/HOL with time [17] incorporates Atkey's [1] idea to include *time credits* in
 353 separation logic into the Imperative/HOL [2] framework. In essence, it enables reasoning
 354 about imperative programs and their run-time in Isabelle/HOL. While all the details can be
 355 found in Section 2.1 of [17], we will give an abstract explanation here that suffices for our
 356 purposes.

A procedure in the monad takes a heap as input and can either fail or return a tuple consisting of a return value, a new heap and a natural number, specifying the number of computation steps used. The type of a procedure with result type $'a$ is given by:

```
datatype 'a Heap = Heap (heap  $\Rightarrow$  ('a  $\times$  heap  $\times$  nat) option)
```

The bind operator as well as fix point iteration, while and other combinators are defined in a straightforward manner. The term $(h, c) \Rightarrow (r, h', t)$ expresses that procedure c started on heap h does not fail and takes time t to produce result r and heap h' .

While heaps themselves do not form a separation algebra, there is an abstraction function α that maps a pair of heap and time credits to an abstract heap. Abstract heaps together with suitable definitions of disjointness and heap addition form a separation algebra. An assertion P , i.e. a mapping from an abstract heap to bool, being true for a heap h and time credits n is denoted by $\alpha(h, n) \models P$. There are basic assertions for an abstract heap containing an array without time credits ($a \mapsto_a xs$), references without time credits ($r \mapsto_r v$) and time credits ($\$n$).

The *separating conjunction* $P * Q$ of two assertions is defined in a straightforward manner, separating both the heap locations and the time credits.

Hoare triples are defined in the following way:

```
<P> c <\lambda r. Q r>_t =
( $\forall h n. \alpha(h, n) \models P \longrightarrow (\exists h' t r. (c, h) \Rightarrow (r, h', t)
\wedge \alpha(h', n - t) \models Q r * true \wedge t \leq n)$ )
```

where the assertion *true* is true for any heap, thus enabling garbage collection of heap elements and time credits. The Hoare triple $\langle P \rangle c \langle \lambda r. Q r \rangle_t$ denotes that procedure c started from a heap satisfying P terminates with a return value r in a resulting heap that satisfies $Q r * true$. In particular it states that the starting heap holds enough time credits n in order to pay for the cost t of executing the procedure c (see line 3).

The cost model assigns most basic commands (e.g. accessing or updating a reference, getting the length of an array) to consume one unit of computation time. Commands that operate on an entire array take $n+1$ units of computation, where n is the length of the array. Examples for basic commands are:

```
<a \mapsto_a xs * $1 * \uparrow(i < |xs|)> Array.upd i x a <\lambda r. a \mapsto_a xs[i:=x] * \uparrow(r = a)>_t
<\$(n+1)> Array.new n x <\lambda r. r \mapsto_a replicate n x>_t
```

where $\uparrow P$ is a pure assertion, which is valid for an empty heap if P holds globally, $xs[i:=x]$ denotes a list xs updated at position i with value x , and $replicate n x$ denotes a list of n elements x .

In Section 4 we review available and new infrastructure and automation for proving valid Hoare triples of procedures in the time-aware monad of Imperative/HOL.

3.2 Generic Sepref

As a next step we want to automatically synthesize programs in the time-aware Imperative/HOL monad from abstract algorithms in the NREST monad. This step is performed by an adaptation of the Sepref tool [12]. The core of the tool is the *translation phase*, where the concrete program is synthesized. We focus on that phase as the other phases can be adapted in a straightforward manner.

The translation works by symbolically executing the abstract program, thereby synthesizing a structurally similar concrete program. During the symbolic execution, the relation

between the abstract and concrete variables is modeled by refinement assertions. The *synthesis predicate* $hnr \Gamma m_{\dagger} \Gamma' R m$ means that concrete program m_{\dagger} implements abstract program m , where Γ contains the refinements for the variables before the execution, Γ' contains the refinements after the execution, and R is the refinement assertion for the result of m . For example, a `bind` is processed by the following synthesis rule:

$$\begin{array}{l} 412 \quad hnr \Gamma m_{\dagger} \Gamma' R_x m \wedge \\ 413 \quad (\forall x x_{\dagger}. hnr (R_x x x_{\dagger} * \Gamma') (f_{\dagger} x_{\dagger}) (R'_x x x_{\dagger} * \Gamma'') R_y (f x)) \\ 414 \quad \longrightarrow hnr \Gamma (\text{do } \{x_{\dagger} \leftarrow m_{\dagger}; f_{\dagger} x_{\dagger}\}) \Gamma'' R_y (\text{do } \{x \leftarrow m; f x\}) \end{array}$$

To refine $x \leftarrow m; f x$, we first execute m , synthesizing the concrete program m_{\dagger} (line 1). The state after m is $R_x x x_{\dagger} * \Gamma'$, where x is the result created by m . From this state, we execute $f x$ (line 2). The new state is $R'_x x x_{\dagger} * \Gamma'' * R_y y y_{\dagger}$, where y is the result of $f x$.

While executing the abstract program, not only a concrete program is created, but also the set of refinement assertions Γ evolves: It contains all the data structures (pure or on the heap) that the concrete program maintains.

All the other combinators (*RECT*, *while*, *if*, *case* ...) have similar rules that are used to decompose an abstract program into parts, synthesize corresponding concrete parts recursively and combine them afterwards.

At the leaves of this decomposition one has to find “atomic” operations, with a suitable synthesis rule. An example could be the rule for the specification of the compare operation of a disjoint-sets data structure as in the concrete Kruskal program in Figure 1b:

$$\begin{array}{l} 429 \quad hnr (is_uf R' R * nat_assn a' a * nat_assn b' b) (uf_cmp R a b) \\ 430 \quad (is_uf R' R * nat_assn a' a * nat_assn b' b) \\ 431 \quad bool_assn (RES [djs_cmp R' a' b' \mapsto itt]) \end{array}$$

The program uf_cmp in the time-aware Imperative/HOL monad refines the abstract compare operation djs_cmp . If the parameters fulfill the correct refinement assertions, i.e. R is a concrete union-find implementation of the abstract equivalence relation R' , as well as $a' = a$ and $b' = b$, then the result of the concrete operation is equal ($bool_assn$) to the result of the abstract one, and the parameters are still in the refinement relations as before.

3.3 Heap-monad to Nondeterminism Refinement (HNR)

Now we present how we can link NREST with the Imperative/HOL monad via a suitable synthesis predicate.

$$\begin{array}{l} 442 \quad hnr \Gamma c \Gamma' R m \equiv m \neq FAIL \longrightarrow \\ 443 \quad (\forall h n. \alpha(h, n) \models \Gamma \longrightarrow (\exists h' t r. (c, h) \Rightarrow (r, h', t) \\ 444 \quad \wedge (\exists t_a r_a. \alpha(h', (n+t_a)-t) \models \Gamma' * R r_a r * true \\ 445 \quad \wedge \text{consume} (\text{return } r_a) t_a \leq m \wedge n+t_a \geq t))) \end{array}$$

If the abstract program m does not fail, procedure c started from a heap satisfying Γ produces a heap satisfying Γ' and a result r which relates to an abstract result r_a via relation R . The abstract result r_a is a valid result of m and has at least t_a time units reserved for it. Together with the time credits on the heap n this pays for the execution cost t (line 4).

In particular, the execution cost t is paid for by the time units t_a specified by the abstract program and by time credits n that are hidden in the data structures on the heap. One can see, that amortized data structures seamlessly integrate into the framework: only amortized run-time costs are visible to the abstract algorithm, while the actual run-time and potential is hidden in the implementation.

457 In order to verify that this definition makes sense, observe what we can prove for it:
 458 First, this definition enables us to prove soundness of the synthesis rule for `bind` from above.
 459 Second, as a final step in an algorithm analysis we would like to extract a Hoare triple for
 460 the concrete program we synthesized. The run-time of final algorithms that we analyse is
 461 typically not dependent on the result, but only on the input. For programs with specifications
 462 of that special form $SPEC\ P(\lambda_ . t)$ we can extract a standard Hoare triple from a valid
 463 synthesis predicate and vice versa:

$$464 \text{ hnr } \Gamma\ c\ \Gamma'\ R\ (SPEC\ P(\lambda_ . t)) \longleftrightarrow \langle \Gamma * \$\ t \rangle\ c\ \langle \lambda r. \Gamma' * (\exists_A\ r'. R\ r'\ r * \uparrow(P\ r')) \rangle_t$$

467 While during reasoning the abstract time bound needs to depend on the result (in order
 468 to prove the synthesis rule for `bind` correct), when proving the run-time of an algorithm, in
 469 most cases the final run-time only depends on the input parameters.

470 Based on that definition we can provide sound synthesis rules for all the combinators
 471 (`bind`, `return`, `while`, `RECT`), as well as a frame and a consequence rule. To illustrate that
 472 the hnr-approach allows to use amortized data structures seamlessly, consider the example
 473 on usage of dynamic arrays in Appendix A.

474 4 Modular Algorithms and Proof Development

475 Using our methodology, algorithm design and analysis can be modularized in two ways:

476 First, separating the implementation details of data structures from the abstract arguments
 477 of algorithms enables focusing on one part of the problem at a time. Both levels have their
 478 own language (time-aware Imperative/HOL and the NREST monad), and the interface is
 479 realized by abstract operations (e.g. `mop_append_list`) and `hnr` rules. Sepref is employed to
 480 automatically synthesize concrete algorithms from abstract ones. On the abstract level we
 481 reserve some amount of time for each abstract operation, whose details will get filled in once
 482 one decides which data structure and concrete operation to use, then yielding a sound upper
 483 bound on the run-time. A collection of abstract operations and their implementations by
 484 efficient data structures will be given in the next subsection.

485 Second, the refinement calculus of NREST programs enables to formulate abstract
 486 algorithms that can be reused as components in larger developments. One example is a
 487 generic BFS component, that is used as a sub-component in the Edmonds–Karp algorithm.
 488 Also abstract algorithms, such as the minimum weight basis algorithm can be formulated on
 489 general matroids, and then later be instantiated for the cycle matroid yielding a blue-print
 490 for Kruskal’s algorithm.

491 Library of Operations and Algorithms

492 Table 1 lists abstract data structures with their abstract operations and the implementations
 493 we currently provide in the *Timed Imperative Isabelle Collections Framework* (TIICF). Note:
 494 it is easy to extend this list. As an example for a generic re-usable algorithm we provide
 495 breadth first search, which is used in the formalization of the Edmonds–Karp algorithm.

496 Methodology and Automation

497 The process of formalizing an algorithm is supported by automation in four stages. We
 498 present those from the most abstract to the concrete:

■ **Table 1** This table shows the abstract data structures with abstract operations that we provide implementations for in the THCF. Amortized run-time bounds are marked with an asterisk (*).

abstract	operations	run-time	concrete
matrix	create; lookup, update	$O(n^2); O(1)$	array
set/map	create; insert, lookup, delete, update	$O(1); O(\log n)$	red-black tree
		$O(n); O(1)$	array
list	create, append; lookup, update	$O(1)^*; O(1)$	dynamic array
disjoint sets	create; union, find	$O(n); O(\log n)$	union-find

499 First, when proving the refinement of a specification in the NREST monad to an abstract
 500 algorithm the generation of verification conditions is automated. They can be discharged by
 501 automatic tactics or interactive proof.

502 Second, abstract algorithms are refined to structurally similar concrete algorithms. Here
 503 a lock step simulation is carried out automatically by the refinement condition generator.
 504 An example is to show the refinement between the programs in Figure 1.

505 Third, the adapted Sepref tool automatically synthesizes a program in the time-aware
 506 Imperative/HOL monad from a given abstract algorithm containing only abstract operations
 507 with available *hnr* rules. Automatic proving of side-conditions is performed in a limited
 508 way. Usually, preconditions of concrete operations are provided as an **assert** in the abstract
 509 algorithm.

510 Finally, for showing that concrete implementations of abstract operations are correct and
 511 satisfy the given time bounds one has to show *hnr* predicates. In essence, these are Hoare
 512 triples in time-aware Imperative/HOL. Zhan et al. [17] develop a methodology for proving
 513 functional correctness and (amortized) run-time claims and provide a setup for automation.
 514 One novel component is a special routine for handling time credits during frame inference.
 515 Lammich [12] provides *sep_auto* — a strong automation for vanilla Imperative/HOL — which
 516 we extend by the above mentioned time frame inference routine to also handle programs in
 517 the time-aware case. Both approaches can be used in order to establish correct Hoare triples
 518 of basic data structures and form a library of algorithms and data structures which can be
 519 used as abstract operations in more advanced algorithms.

520 5 Case-Studies

521 Besides smaller examples (remove duplicates with set implementation using red-black trees,
 522 binary search, and Floyd–Warshall) we provide verification of functional correctness and
 523 time bounds of Kruskal’s algorithm and the Edmonds–Karp algorithm.

524 Kruskal

525 Kruskal’s algorithm was verified in the standard Refinement Framework in parallel to the
 526 research reported on in this paper. It can be found in the archive of formal proofs [9]. As a
 527 case study, we port it to NREST, adding the run-time claims.

528 The proof development follows this general structure: first we define the abstract algorithm
 529 for minimum weight basis in matroids (c.f. Figure 1a) and verify it. Then we instantiate
 530 it with the cycle matroid for forests in undirected graphs and refine the algorithm with
 531 the usage of equivalence classes. Figure 1b shows the last-but-one stage in the step-wise
 532 refinement process. In a last step we fix the vertices to be natural numbers and the domain

533 of the disjoint-set data structure to be the set from $\{0, \dots, M\}$, with M being the maximal
534 vertex in the graph. After that, we use the implementation of the union-find data structure
535 from the TIICF to synthesize a concrete algorithm with the Sepref tool.

536 Provided a procedure that obtains a list of edges of a graph in linear time, a $O(n * \log n)$
537 sorting algorithm and a union-find data structure with logarithmic find and union operations
538 we obtain a concrete algorithm that calculates the minimum weight spanning forest for the
539 graph in time $O(E * \log E + M + E * \log M)$, with E being the number of edges and M being
540 the maximal vertex in the graph.

541 We have only proven the logarithmic bounds for the union-find data structure for this
542 case-study. Chaguéraud et al. [4] verified a union-find data structure with amortized
543 run-time $O(\alpha(M))$ (where M is the size of the domain of the disjoint-set data structure and
544 α is the inverse Ackermann function) in Coq.

545 When developing this case study, we learned that the correctness arguments can be
546 plainly reused, and that adding the proofs of the run-time claims does not interfere. However,
547 it is necessary to add more assertions in the algorithms that speak about the sizes of the data
548 structures used. This reasoning is mostly done on the abstract level, but the information has
549 to be passed to the concrete algorithm via assertions. In the Sepref translation phase, this
550 information is needed to discharge the preconditions of the *hnr* predicates, which demand
551 that enough time has been reserved to execute the step.

552 Edmonds–Karp: Reuse

553 Before starting this project we had the following working hypothesis:

554 “Formalizations in the standard Refinement Framework can be easily extended to also
555 verify the run-time behaviour. In this process, most of the formalization can be reused, and
556 termination arguments can be translated into run-time arguments.”

557 We conducted this extension to the Edmonds–Karp algorithm [13, 14] as a case-study.
558 The result is two-fold: For procedures where the reasoning on the run-time of the algorithm
559 is already well prepared making this claim explicit is straightforward, for procedures where
560 only termination has been shown only coarse bounds can be shown with little effort. Fine
561 tuned run-time bounds require substantial work.

562 Lammich et al. [14] already quite explicitly work out the run-time bound for the outer
563 loop of the Edmonds–Karp algorithm. We were able to reuse the whole proof and additionally
564 embed the result into our time aware non-determinism monad, thus making the run-time
565 claim less ad-hoc. On the other hand the inner breadth-first search is only proven to terminate
566 via a terminating lexicographic ordering. Plainly using this leads to a valid but very coarse
567 run-time bound. Establishing the tight $O(E + V)$ bound involves some amortized argument
568 on the abstract level and was a considerable verification effort, but again orthogonal to the
569 functional correctness proof, which in turn can be reused with no change.

570 As this case study uses matrices to represent the residual networks, we had to rework
571 the array implementation of matrices for the TIICF, with linear-time initialization, and
572 constant-time update and lookup operations.

573 **6** Conclusion

574 Related Work

575 Lammich pioneered the Sepref tool [12] and it has been used to verify several interesting
576 algorithms and software projects [13, 6, 16]. It was recently adapted to synthesize programs

577 in LLVM [10] instead of Imperative/HOL. Coming up with a generic Sepref tool that is
 578 parametrized in the target and source language, as well as extending the LLVM semantics to
 579 run-time are interesting future projects.

580 As already mentioned, time-aware Imperative/HOL is due to Zhan et al. [17], which
 581 builds upon Atkey’s [1] idea to use Time Credits in Separation Logic.

582 In the Coq community similar theory [3, 7] and the run-time analysis of interesting
 583 algorithms [8] and data structures [4] have been formalized.

584 To the best of our knowledge, we are the first to combine run-time analysis with refinement.

585 Limitations and Future Work

586 In particular, we are not satisfied with the parametrization of operations with timing functions.
 587 We envision not only counting one currency (\$) representing one computation step in the
 588 final concrete algorithm, but to have currencies for abstract operations. Say one abstract
 589 algorithm A incurs cost of one “ A -dollar” $\$A$ and can be implemented by an algorithm using
 590 several operations C_1 and C_2 costing some $\$C_1$ and some $\$C_2$. Refining algorithms that use
 591 several calls to A should then routinely yield a refinement with costs in terms of $\$C_1$ and
 592 $\$C_2$. A target monad of Sepref then would also differentiate between several actions and
 593 respective currencies of costs. Refining abstract operations into this target would exchange
 594 these currencies in a sound way, such that ultimately upper bounds on the usage of these
 595 currencies are obtained.

596 In this paper we only study upper bounds of run-time of algorithms. This should be
 597 relaxed in at least two ways: First, while run-time is a particularly interesting case, also other
 598 quantities are possible, e.g. stack usage, or energy usage. Second, not only upper bounds can
 599 be reasoned about, also lower bounds are feasible. A refinement relation on lower bounds
 600 seems to be straightforward. Also combining this in a pair of *enats* and keeping track of
 601 lower as well as upper bounds seems to be feasible.

602 We already mentioned, that Lammich’s LLVM semantics could be extended to counting
 603 the number of operations.

604 Obviously, it is future work to further fill the collection of efficient data structures and
 605 reusable algorithms, as well as lowering the barriers to verify run-time arguments by providing
 606 more automation.

607 Conclusion

608 In this paper, we have combined the refinement approach of algorithm verification with
 609 techniques to verify the run-time of algorithms: We extended the Isabelle Refinement
 610 Framework to express the result and time consumption of abstract algorithms as well as the
 611 Sepref tool to synthesize executable imperative programs for such abstract algorithms. This
 612 setup makes it possible to carry out the verification of algorithms such as Edmonds–Karp
 613 and Kruskal in a modular way. Separating concerns into the abstract algorithmic idea and
 614 the implementation details of data structures makes larger proof developments feasible.

615 Our use-cases indicate that for additionally verifying run-time arguments for algorithms
 616 whose functional correctness has already been shown within the vanilla Isabelle Refinement
 617 Framework, formalizations can be reused to a large extent. We think that even larger
 618 developments can be tackled this way, both verifying functional correctness and the run-time
 619 analysis of such algorithms.

References

- 620 —
- 621 1 Robert Atkey. Amortised resource analysis with separation logic. In *ESOP*, volume 6012,
622 pages 85–103. Springer, 2010.
- 623 2 Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkok, and John Matthews.
624 Imperative functional programming with Isabelle/HOL. *Lecture Notes in Computer Science*,
625 5170:134–149, 2008.
- 626 3 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In
627 *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*,
628 ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM. URL: <http://doi.acm.org/10.1145/2034773.2034828>, doi:10.1145/2034773.2034828.
- 629 4 Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity
630 of a union-find implementation in separation logic with time credits. *Journal of Automated*
631 *Reasoning*, pages 1–35, 2017.
- 632 5 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and
633 Jan-Georg Smaus. A fully verified executable ltl model checker. In *International Conference*
634 *on Computer Aided Verification*, pages 463–478. Springer, 2013.
- 635 6 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified sat solver with
636 watched literals using imperative hol. In *Proceedings of the 7th ACM SIGPLAN International*
637 *Conference on Certified Programs and Proofs*, pages 158–171. ACM, 2018.
- 638 7 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing
639 asymptotic complexity claims via deductive program verification. In *European Symposium on*
640 *Programming (ESOP)*, 2018.
- 641 8 Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pot-
642 tier. Formal proof and analysis of an incremental cycle detection algorithm. Submitted,
643 February 2019. URL: <http://gallium.inria.fr/~agueneau/publis/gueneau-jourdan-chargueraud-pottier-2019.pdf>.
- 644 9 Maximilian P.L. Haslbeck, Peter Lammich, and Julian Biendarra. Kruskal’s algorithm for
645 minimum spanning forest. *Archive of Formal Proofs*, February 2019. <http://isa-afp.org/entries/Kruskal.html>, Formal proof development.
- 646 10 Peter Lammich. Generating verified LLVM from Isabelle/HOL. submitted.
- 647 11 Peter Lammich. Verified efficient implementation of gabow’s strongly connected component
648 algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340.
649 Springer, 2014.
- 650 12 Peter Lammich. Refinement to Imperative/HOL. In *International Conference on Interactive*
651 *Theorem Proving*, pages 253–269. Springer, 2015.
- 652 13 Peter Lammich and S Reza Sefidgar. Formalizing the edmonds-karp algorithm. In *International*
653 *Conference on Interactive Theorem Proving*, pages 219–234. Springer, 2016.
- 654 14 Peter Lammich and S. Reza Sefidgar. Formalizing the edmonds-karp algorithm. *Archive*
655 *of Formal Proofs*, August 2016. http://isa-afp.org/entries/EdmondsKarp_Maxflow.html,
656 Formal proof development.
- 657 15 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to
658 Hopcroft’s algorithm. In *International Conference on Interactive Theorem Proving*, pages
659 166–182. Springer, 2012.
- 660 16 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *Inter-*
661 *national Conference on Tools and Algorithms for the Construction and Analysis of Systems*,
662 pages 61–78. Springer, 2018.
- 663 17 Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative
664 programs in Isabelle. In *International Joint Conference on Automated Reasoning*, pages 532–
665 548. Springer, 2018.
- 666
- 667
- 668

669 **A** Example: Amortized Dynamic Array and Remove Duplicates

670 Let us consider an abstract operation and its implementation: appending an element to the
671 back of a list.

672 $mop_push_list\ t\ x\ xs = RES\ [xs \cdot [x] \mapsto t\ xs]$
673
674

675 The operation is specified in the NREST monad, with a new parameter t that represents
676 the run-time of the operation, here parametrized in the list xs . For an implementor, this
677 leaves open the possibility to later provide an implementation whose time consumption
678 depends on xs , e.g. on its length. Let us turn to an implementation of that operation on a
679 dynamic array.

680 Implementation

681 An *abstract dynamic list* is represented by a pair of a carrier list bs and a fill level n . The
682 corresponding abstract list as is the list bs restricted to the first n elements:

683 $dyn_abs\ (bs, n)\ as \iff as = take\ n\ bs \wedge n < |bs|$
684
685

686 We define a function $push_array_fun$ on abstract dynamic lists that doubles the length
687 of the list if it is full and then appends an element. We prove its functional correctness:

688 $dyn_abs\ (bs, n)\ as \longrightarrow dyn_abs\ (push_array_fun\ x\ (bs, n))\ (as \cdot [x])$
689
690

691 Recall that $p \mapsto_a xs$ denotes a heap containing an array at address p with content xs .
692 Based on this, one can define an assertion

693 $dyn_array_raw\ (bs, n)\ (p, m) = (p \mapsto_a bs * \uparrow(m = n))$
694
695

696 relating an abstract dynamic list with a concrete *dynamic array* represented by a pair of
697 address p and fill level m .

698 For the functional $push_array_fun$ we define a corresponding procedure $push_array$
699 which appends an element to the back of a dynamic array, doubling the length if it is
700 exceeded. We can now show the following raw Hoare triple, with worst-case run-time linear
701 in the fill level of the dynamic array, as we might have to double the array. The explicit
702 numbers in the run-time stem from the concrete implementation of $push_array$ and the cost
703 model of time-aware Imperative/HOL.

704 $n \leq length\ bs \longrightarrow$
705 $\langle dyn_array_raw\ (bs, n)\ p * \$(5*n + 9) \rangle$
706 $push_array\ x\ p$
707 $\langle \lambda p'. dyn_array_raw\ (push_array_fun\ x\ (bs, n))\ p' \rangle_t$
708
709

710 We now incorporate the potential $(\Phi(bs, n) = 10 * n - 5 * |bs|)$ into an assertion for a
711 compound data structure dyn_array and prove the following Hoare triple with amortized
712 constant run-time:

713 $dyn_array\ r\ p = dyn_array_raw\ r\ p * \$(\Phi\ r)$
714
715

716 $n \leq length\ bs \longrightarrow$
717 $\langle dyn_array\ (bs, n)\ p * \$19 \rangle$
718 $push_array\ x\ p$
719 $\langle \lambda p'. dyn_array\ (push_array_fun\ x\ (bs, n))\ p' \rangle_t$
720
721

722 Note that for showing the latter amortized Hoare triple it does not suffice to employ the
723 raw Hoare triple, rather *push_array* must be unfolded again.

724 As a final step we compose the refinements of abstract lists to abstract dynamic lists
725 (*dyn_abs*) and further to dynamic arrays (*dyn_array*) and obtain *dyna_assn*:

$$726 \text{dyna_assn } as \ p = (\exists_A bs \ n. \text{dyn_array } (bs, n) \ p * \uparrow(\text{dyn_abs } (bs, n) \ as))$$

729 where the list and fill level of the abstract dynamic array are hidden behind an existential
730 quantifier. Then we obtain the final Hoare triple of the procedure:

$$731 \langle \text{dyna_assn } as \ p * \$19 \rangle \text{push_array } x \ p \ \langle \lambda p'. \text{dyna_assn } (as \cdot [x]) \ p' \rangle_t$$

734 Together with the definition of *mop_push_list* we can state and prove the synthesis
735 predicate for the append operation:

$$736 \ 19 \leq t \ xs' \longrightarrow \text{hnr } (\text{dyna_assn } xs' \ p * \text{Id } x' \ x) (\text{push_array } x \ p) \\ 737 \ (\text{Id } x' \ x) \ \text{dyna_assn } (\text{mop_push_list } t \ x' \ xs')$$

740 Usage

741 The abstract operation *mop_push_list* can now be used when specifying an abstract algorithm.
742 Then a concrete time function *t* can be specified, which is used to determine the overall cost
743 of the algorithm.

744 Consider the following program to remove duplicates from a list.

```
745 remdups_impl as = do {
746   ys ← mop_empty_list 12;
747   S ← mop_set_empty 1;
748   (zs, ys, S) ← whileT (λ(xs, ys, S). |xs| > 0) (λ(xs, ys, S). do {
749     assert (|xs| > 0 ∧ |xs| + |ys| ≤ |as| ∧ |S| ≤ |ys|);
750     (x, xs) ← return (hd xs, tl xs);
751     b ← mop_set_member (λ_. rbt_search_t (|as| + 1) + 1) x S;
752     if b then
753       return (xs, ys, S)
754     else do {
755       S ← mop_set_insert (λ_. rbt_insert_t (|as| + 1)) x S;
756       ys ← mop_push_list (λ_. 23) x ys;
757       return (xs, ys, S)
758     }
759   }) (as, ys, S);
760   return ys
761 }
```

764 The program uses *mop_push_list* from above as well as other abstract operations. For
765 example insertion into a set:

$$766 \text{mop_set_insert } t \ x \ S = \text{RES } [S \cup \{x\} \mapsto t \ S]$$

769 Let *remdups_t* *n* = *n**(60 + *rbt_search_t* (*n*+1) + *rbt_insert_t* (*n*+1)) + 20, then our
770 automation can prove the following refinement theorem and asymptotic bound:

$$771 \text{remdups_impl } as \leq \text{SPEC } (\lambda ys. \text{set } ys = \text{set } as \wedge \text{distinct } ys) (\lambda_. \text{remdups_t } |as|) \\ 772 \text{remdups_t} \in \Theta(\lambda n. n * \log n)$$

23:18 Refinement with Time

775 For each operation in the program some time is reserved. The overall run-time of the
776 program is then a function of these reserved quantities. Note that for the set operations we
777 reserved time not parametrized in the size of the set it operates on, but in the length of the
778 input list as , which is an upper bound.

779 When synthesizing an Imperative/HOL program, the synthesis rules will be applied and
780 their preconditions must be discharged. For the mop_push_list this boils down to the trivial
781 check $16 \leq 23$. Note that in that process only the advertised cost of the dynamic array is
782 concerned, while the amortization is hidden on this level.

783 Let us consider a more interesting operation. The synthesis rule of the red-black tree
784 implementation of mop_insert_set is the following:

785 $rbt_insert_t (card\ S + 1) \leq t\ S \longrightarrow$
786 $hnr (Id\ x' x * rbt_set_assn\ S\ p) (rbt_set_insert\ x\ p)$
787 $(Id\ x' x) rbt_set_assn (mop_set_insert\ t\ x' S)$
788
789

790 where $rbt_set_assn\ S\ p$ relates a set S with a red-black tree at address p . During synthesis
791 the Sepref tool has to check whether there is enough reserved time for the set insertion.

792 $|S| \leq |ys| \wedge |xs| + |ys| \leq |as|$
793 $\longrightarrow rbt_insert_t (|S| + 1) \leq (\lambda_. rbt_insert_t (|as| + 1))\ S$
794
795

796 The goal can be discharged with the knowledge from the assertions and the monotony of
797 rbt_insert_t .

798 Once more, note that amortized data structures seamlessly can be modeled using time
799 credits, and this comfort can be further extended to also be available for the abstract
800 algorithm. At the abstract level, an amortized data structure behaves just as a normal data
801 structure does.