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

Maximilian P. L. Haslbeck
Technische Universität München, Germany

Peter Lammich
The University of Manchester, England

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 for the first time, we present a methodology for verifying the functional correctness and the run-time analysis of algorithms in a modular way. We use it to 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 [11], 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.ITP.2019.20

Supplement Material The formalization described in this paper is available at `https://www21.in.tum.de/~haslbema/Sepreftime`.

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, as well as the anonymous reviewers 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 [10, 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

use step-wise refinement to form concrete algorithms. As a last step the Sepref tool [11] can be used to synthesize efficient executable imperative code while preserving correctness. Target languages for that tool are hybrid languages such as SML, Scala and more recently the purely imperative language LLVM [12]. Such verification efforts result in executable algorithms that are often competitive with real world implementations within one order of magnitude. However, only functional correctness is ensured and not run-time bounds.

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

Our vision is to specify abstract algorithms and their run-time in terms of abstract operations with time bounds – say Edmonds–Karp algorithm uses at most $E * V$ find-augmenting-path operations. When we then refine an operation like find-augmenting-path to a more concrete BFS algorithm involving operations such as set membership test and map lookup, we can also refine the abstract compound algorithm to use the more refined operations. Just as for plain refinement we separate abstract run-time arguments from reasoning about run-times of concrete data structures. As a last step we synthesize executable imperative code which refines the abstract algorithm and thus obeys both the high-level correctness theorem and the run-time bound. This synthesis step is only successful when meaningful time bounds have been specified. For abstract programs with absurd run-time bounds it will just not be possible to synthesize real programs.

The main contributions of this paper are:

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

2 Non-determinism Monad With Time

In this section we introduce NREST, the timed non-determinism monad. It allows specifying the result and time consumption of programs. As this is an extension of the NRES monad of the Isabelle Refinement Framework, we follow Lammich [11] in some of our explanations.

2.1 Timed Non-determinism Monad

We want to specify the result of a computation together with its worst case execution time. We design the monad to permit three monadic effects: First, we allow non-deterministic selection from a set of computation results. This is a common technique in program refinement, used to hide implementation details of abstract algorithms. Second, we support failure in order to model non-termination and assertions. Last, we model upper bounds on the run time for each possible result.[1] A program in the timed non-determinism monad is defined over the type α *NREST*:

[1] Alternatively, one could use a set of pairs of result and time constraint. However, this would not be *fully abstract* wrt. upper bounds, in the sense that $\{(r,1),(r,3)\}$ would be equivalent to $\{(r,3)\}$.

α *NREST* = *RES* ($\alpha \Rightarrow$ *enat option*) | *FAIL*,

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

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

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

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

On the type NREST we define the following functions:

```
consume :: α NREST ⇒ enat ⇒ α NREST where
consume (RES M) t = RES (λx. case M x of None ⇒ None | Some t' ⇒ Some (t + t'))
consume FAIL t = FAIL
```

```
return :: α ⇒ α NREST where
return x = RES [ x ↦ 0 ]
```

```
bind :: α NREST ⇒ (α ⇒ β NREST) ⇒ β NREST  where
bind (RES M) f = Sup { consume (f x) t |x t. M x = Some t}
bind FAIL f = FAIL
```

The term `consume` M t describes the computation M prolonged by t time steps, `return` x is a computation that yields a single result x in no time, and `bind` m f is the sequential composition of two computations: First compute any result x of m, then any result y of $f\ x$. The time bounds for the final results have to be determined considering all possible ways how to reach them. If m or any reachable computation path of f fails the compound computation also fails. NREST together with `bind` and `return` forms a monad and `bind` as well as `consume` are monotonic w.r.t. the refinement ordering:

```
m ≤ m' ⟶ (∀x. f x ≤ f' x) ⟶ bind m f ≤ bind m' f'
m ≤ m' ⟶ t ≤ t' ⟶ consume m t ≤ consume m' t'
```

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

Furthermore we define two derived operations:

$SPEC :: (\alpha \Rightarrow bool) \Rightarrow (\alpha \Rightarrow enat) \Rightarrow \alpha\ NREST$ **where**
$SPEC\ P\ t = RES\ (\lambda v.\ \texttt{if}\ P\ v\ \texttt{then}\ Some\ (t\ v)\ \texttt{else}\ None)$

$\texttt{assert} :: bool \Rightarrow unit\ NREST$ **where**
$\texttt{assert}\ P = (\texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else}\ FAIL)$

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

$P \longrightarrow m \leq m' \longrightarrow \texttt{do}\ \{\ \texttt{assert}\ P;\ m\ \} \leq m'$
$(P \longrightarrow m \leq m') \longrightarrow m \leq \texttt{do}\ \{\ \texttt{assert}\ P;\ m'\ \}$

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

2.2 Recursive Programs

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

$RECT :: ((\beta \Rightarrow \alpha\ NREST) \Rightarrow \beta \Rightarrow \alpha\ NREST) \Rightarrow \beta \Rightarrow \alpha\ NREST$ **where**
$RECT\ F\ x = (\texttt{if}\ mono2\ F\ \texttt{then}\ (gfp\ F\ x)\ \texttt{else}\ FAIL)$

Here, *mono2* denotes monotonicity w.r.t. both the flat ordering and the refinement ordering. The benefits of this are explained in more detail elsewhere [11]. Note that programs constructed by the combinators we introduced above are monotonic in that sense by construction. The combinator *RECT* is also monotonic w.r.t. the refinement ordering:

$mono2\ B \wedge (\forall F\ x.\ B\ F\ x \leq B'\ F\ x) \longrightarrow RECT\ B\ x \leq RECT\ B'\ x$

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

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

```
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 }
```

(a) The greedy algorithm to construct a minimum weight basis of a Matroid in the NREST monad.

```
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 }
```

(b) A further refinement for the Kruskal algorithm, where an additional disjoint sets data structure is passed around.

Figure 1 Two examples of algorithms in the the timed non-determinism monad.

In line two the algorithm obtains a list of the elements of the carrier set E (later this will be the set of edges of an undirected graph) sorted w.r.t. some weight function w. Starting from an empty independent set, we iteratively add elements if they leave the set T independent (i.e. create no cycle in the graph case). For all operations that may cost time, we reserve some time parameter of type *nat* or functions to *nat*: here t_{sc}, t_{eb}, t_{it} and t_i stand for sorted carrier set time, empty basis time, independence test time and insertion time.

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

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

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

2.3 Generalizing the Weakest Precondition

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

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

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

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

$$c \leq RES\ Q \longleftrightarrow Some\ 0 \leq lst\ c\ Q$$

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

$$lst\ (\texttt{bind}\ m\ f)\ Q = lst\ m\ (\lambda y.\ lst\ (f\ y)\ Q)$$

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

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

$$lst\ c\ Q = Inf\ r.\ minus\ Q\ c\ r$$

Operation $minus :: (\alpha \Rightarrow enat\ option) \Rightarrow \alpha\ NREST \Rightarrow \alpha \Rightarrow enat\ option$ formalizes taking the difference. We have the following cases:

- c fails: then c may never be executed and thus there is no valid latest starting time, i.e. $minus\ Q\ c\ r = None$.
- $c{=}RES\ C$ and $C\ r = None$: as C will never produce the result r, it can be ignored, i.e. the result is the top element: $Some\ \infty$.
- $c{=}RES\ C$ and $C\ r = Some\ m$ and $Q\ r = None$: r is specified to not be obtained, but when starting c we obtain r, thus there is no valid starting time for C: $minus\ Q\ c\ r = None$.
- $c{=}RES\ C$ and $C\ r = Some\ m$ and $Q\ r = Some\ n$: if more time is needed than specified ($n < m$) there is no valid latest starting time and we return $None$, otherwise the difference is returned ($Some\ (n{-}m)$).

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

$$\begin{aligned}
& c \leq RES\ Q \\
\longleftrightarrow\ & Some\ 0 \leq lst\ c\ Q && (= Inf\ r.\ \ minus\ Q\ c\ r\ ) \\
\longleftrightarrow\ & \forall r.\ \ Some\ 0 \leq minus\ Q\ c\ r
\end{aligned}$$

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

2.4 Sound proof rules for the latest starting time calculus

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

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

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

For the other combinators we have:

$$\begin{aligned}
&(\forall r{\in}M.\ Some\ (t + M\ r) \leq Q\ r) \longrightarrow Some\ t \leq lst\ (RES\ M)\ Q \\
&Some\ t \leq Q\ x \longrightarrow Some\ t \leq lst\ (\texttt{return}\ x)\ Q \\
&(\forall x.\ P\ x \longrightarrow Some\ (t + t'\ x) \leq Q\ x) \longrightarrow Some\ t \leq lst\ (SPEC\ P\ t')\ Q \\
&Some\ (t + t') \leq lst\ M\ Q \longrightarrow Some\ t \leq lst\ (\texttt{consume}\ M\ t')\ Q
\end{aligned}$$

For the fold operation $nfold :: \beta\ list \Rightarrow (\beta \Rightarrow \alpha \Rightarrow \alpha\ NREST) \Rightarrow \alpha \Rightarrow \alpha\ NREST$ we have the following rule:

$$
\begin{aligned}
&I\ []\ l_0\ s_0 \\
&\wedge\ (\forall x\ l_1\ l_2\ s.\ l_0{=}l_1 \cdot [x] \cdot l_2 \wedge I\ l_1\ ([x] \cdot l_2)\ s \\
&\qquad \longrightarrow Some\ 0 \leq lst\ (f\ x\ s)\ (emb\ (I\ (l_1 \cdot [x])\ l_2)\ t_{body})) \\
&\wedge\ (\forall s.\ I\ l_0\ []\ s \longrightarrow Some\ (t + t_{body} * |l_0|) \leq Q\ s) \\
&\longrightarrow Some\ t \leq lst\ (nfold\ l_0\ f\ s_0)\ Q
\end{aligned}
$$

Here, $emb\ P\ t = (\lambda x.\ \texttt{if}\ P\ x\ \texttt{then}\ Some\ t\ \texttt{else}\ None)$, *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 $nfold\ l_0\ f\ s_0$ 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`, we construct a syntax directed verification condition generator that exhaustively applies those rules.

▶ **Example 4.** After annotating the loop in the abstract program from Figure 1b with $body_{time} = t_{it} + t_i$ and a suitable invariant $I = \lambda l_1\ l_2\ T.\ I_{mwb}\ (T,\ set\ l_2)$ (where $I_{mwb}(T, E)$ implies $minBasis\ T$ 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:

$$
\begin{aligned}
&sorted\_wrt\ w\ l \wedge distinct\ l \wedge set\ l = E \wedge l = l_1 \cdot [e] \cdot l_2 \wedge indep\ (T \cup \{e\}) \\
&\wedge\ I_{mwb}(T,\ set\ ([e] \cdot l_2)) \longrightarrow I_{mwb}(T \cup \{e\},\ set\ l_2)
\end{aligned}
$$

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 use a list. We have the following refinement in mind. Given that a list l represents a set T (denoted by $(l,T){\in}list\_set\_rel$), the resulting lists of the program on the left hand side refine the resulting sets produced by the right hand side program:

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

Given a refinement relation R, i.e. a relation that relates concrete elements with abstract elements, the concretization function $\Downarrow$R maps abstract results to concrete results w.r.t. R. Note that, if R is single-valued any concrete result is mapped to at most one abstract result.

$$
\begin{aligned}
&\Downarrow R\ FAIL = FAIL \\
&\Downarrow R\ (RES\ X) = RES\ (\lambda c.\ Sup\ \{X\ a |\ a.\ (c,a){\in}R\})
\end{aligned}
$$

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:

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

Analogous rules can be proven for *RECT*, *nfold*, `assert`, and the other combinators.

2.6 Setting Up a VCG for Refinement

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

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

$Kruskal \leq \Downarrow\ list\_graph\_rel\ minWeightBasis$

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

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

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

3 Refinement to Imperative/HOL with Time

In this section we introduce the time-aware monad of Imperative/HOL [17], which we then use as the target monad of the adapted Sepref tool [11] with NREST as the source monad.

3.1 Imperative/HOL with Time

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

```
datatype α Heap = Heap (heap ⇒ (α × heap × 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* $P{*}Q$ expresses that the heap and time credits can be partitioned into two disjoint parts satisfying assertions P and Q respectively. The strength of separation logic is, that this disjointness enables modular reasoning, which also carries over to reasoning about time credits.

Hoare triples are defined in the following way:

```
1 <P> c <λr. Q r>_t =
2  (∀h n. α (h,n) ⊨ P ⟶ (∃h' t r. (c,h) ⇒ (r, h', t)
3                                  ∧ α (h', n − t) ⊨ Q r ∗ true ∧  t ≤ n) )
```

where the assertion *true* is true for any heap, thus enabling garbage collection of heap elements and time credits. The Hoare triple $<P> c <\lambda r.\ Q\ r>_t$ 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 ↦_a xs ∗ $1 ∗ ↑(i < |xs|)> Array.upd i x a <λr. a ↦_a xs[i:=x] ∗ ↑(r = a)>_t
<$(n+1)> Array.new n x <λr. r ↦_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 4.2 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 Adapted 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 [11]. Note that, the original tool refines NRES to vanilla Imperative/HOL; adapting it includes many but rather straightforward modifications. During that process we identified common patterns and constraints on the source and target monad. It is future work to come up with a generalized Sepref tool. 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* guiding the "Heap-monad to Non-determinism Refinement" is denoted by $hnr\ \Gamma\ m_\dagger\ \Gamma'\ R\ m$: it means that the concrete program $m_\dagger$ implements the 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:

```
1  hnr Γ m† Γ' R_x m ∧
2    (∀x x†. hnr (R_x x x† * Γ') (f† x†) (R'_x x x† * Γ'') R_y (f x))
3  ⟶ hnr Γ (do {x† ← m†; f† x†}) Γ'' R_y (do {x ← m; f x})
```

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:

```
hnr (is_uf R' R * nat_assn a' a * nat_assn b' b) (uf_cmp R a b)
    (is_uf R' R * nat_assn a' a * nat_assn b' b)
    bool_assn (RES [djs_cmp R' a' b' ↦ itt] )
```

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 Non-determinism Refinement (HNR)

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

```
1  hnr Γ c Γ' R m ≡ m ≠ FAIL ⟶
2   (∀h n. α(h, n) ⊨ Γ ⟶ (∃h' t r. (c,h) ⇒ (r, h', t)
3                                ∧ (∃t_a r_a. α(h', (n+t_a)−t) ⊨ Γ' * R r_a r * true
4                                    ∧ consume (return r_a) t_a ≤ m  ∧ n+t_a≥t)))
```

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.

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

$$hnr\ \Gamma\ c\ \Gamma'\ R\ (SPEC\ P\ (\lambda\_.\ t)) \longleftrightarrow <\Gamma\ *\ \$\ t> c <\lambda r.\ \Gamma' * (\exists_A\ r'.\ R\ r'\ r * \uparrow(P\ r'))>_t$$

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

Based on that definition we can provide sound synthesis rules for all the combinators as well as a frame and a consequence rule. To illustrate how the hnr-approach allows to use amortized data structures seamlessly, consider the first case-study in Section 5.1.

4 Modular Algorithms and Proof Development

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

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

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

4.1 Library of Operations and Algorithms

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

Table 1 This table shows the abstract data structures with abstract operations that we provide implementations for in the TIICF. 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

4.2 Methodology and Automation

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

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

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

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

Finally, for showing that concrete implementations of abstract operations are correct and satisfy the given time bounds one has to show *hnr* predicates. In essence, these are Hoare triples in time-aware Imperative/HOL. Zhan et al. [17] develop a methodology for proving functional correctness and (amortized) run-time claims and provide a setup for automation. One novel component is a special routine for handling time credits during frame inference. Lammich [11] provides *sep_auto* – a strong automation for vanilla Imperative/HOL – which we extend by the above mentioned time frame inference routine to also handle programs in the time-aware case. Both approaches can be used in order to establish correct Hoare triples of basic data structures and form a library of algorithms and data structures which can be used as abstract operations in more advanced algorithms.

5 Case Studies

In this section we present three case studies:

The first one considers the abstract operation "appending an element to the end of a list" and illustrates three stages of the verification process: implementing the operation for a concrete data structure in Imperative/HOL with time, designing a synthesis predicate relating the concrete with the abstract operation and using it in an abstract algorithm.

The latter two case studies describe the verification of more involved algorithms where refinement helps structuring the development: Kruskal's minimum spanning tree algorithm and the Edmonds–Karp algorithm for maximum network flow.

5.1 Amortized Dynamic Array and Remove Duplicates

Let us consider the following abstract operation, appending an element to the end of a list:

```
mop_push_list t x xs = RES [xs · [x] ↦ t xs]
```

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

Implementation

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

```
dyn_abs (bs,n) as ⟷ as = take n bs ∧ n<|bs|
```

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

```
dyn_abs (bs,n) as ⟶ dyn_abs (push_array_fun x (bs,n)) (as · [x])
```

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

```
dyn_array_raw (bs, n) (p, m) = (p ↦a bs ∗ ↑(m = n))
```

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

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

```
n ≤ length bs ⟶
<dyn_array_raw (bs, n) p ∗ $(5∗n + 9)>
 push_array x p
<λp'. dyn_array_raw (push_array_fun x (bs, n)) p'>t
```

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

```
dyn_array r p = dyn_array_raw r p ∗ $(Φ r)
```

```
n ≤ length bs ⟶
<dyn_array (bs, n) p ∗ $19>
 push_array x p
<λp'. dyn_array (push_array_fun x (bs, n)) p'>t
```

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

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

```
dyna_assn as p = (∃_A bs n. dyn_array (bs,n) p * ↑(dyn_abs (bs,n) as))
```

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

```
<dyna_assn as p * $19> push_array x p <λp'. dyna_assn (as · [x]) p'>_t
```

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

```
19 ≤ t xs' ⟶ hnr (dyna_assn xs' p * Id x' x) (push_array x p)
                 (Id x' x) dyna_assn (mop_push_list t x' xs')
```

Usage

The abstract operation *mop_push_list* can now be used when specifying an abstract algorithm. Then a concrete time function t can be specified, which is used to determine the overall cost of the algorithm. In this example we choose (*λ_. 23*), which is not a tight bound but enough to later allow synthesizing a concrete program using dynamic arrays.

Consider the following program to remove duplicates from a list.

```
1  remdups_impl as = do {
2    ys ← mop_empty_list 12;
3    S ← mop_set_empty 1;
4    (zs,ys,S) ← whileT (λ(xs,ys,S). |xs| > 0) (λ(xs,ys,S). do {
5      assert (|xs| > 0 ∧ |xs| + |ys| ≤ |as| ∧ |S| ≤ |ys|);
6      (x,xs) ← return (hd xs, tl xs);
7      b ← mop_set_member (λ_. rbt_search_t (|as| + 1) + 1) x S;
8      if b then
9        return (xs,ys,S)
10     else do {
11       S ← mop_set_insert (λ_. rbt_insert_t (|as| + 1)) x S;
12       ys ← mop_push_list (λ_. 23) x ys;
13       return (xs,ys,S)
14     }
15   }) (as,ys,S);
16   return ys
17 }
```

The program uses *mop_push_list* from above as well as other abstract operations with corresponding reserved run-time functions. For example insertion into a set:

```
mop_set_insert t x S =  RES [S ∪ {x} ↦ t S]
```

For each operation in the program some time is reserved. The overall run-time of the program is then a function of these reserved quantities.

Let *remdups_t n = n*(60 + rbt_search_t (n+1) + rbt_insert_t (n+1)) + 20.*

Note that for the set operations the reserved time in *remdups_t* is not parametrized in the size of the set they operate on, but in an over-approximation of it: the length of the input. Our automation can prove the following refinement theorem and asymptotic bound:

$$remdups\_impl\ as \leq SPEC\ (\lambda ys.\ set\ ys = set\ as \wedge distinct\ ys)\ (\lambda\_.\ remdups\_t\ |as|)$$
$$remdups\_t \in \Theta(\lambda n.\ n * \log n)$$

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

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

$$rbt\_insert\_t\ (card\ S + 1) \leq t\ S \longrightarrow$$
$$hnr\ (Id\ x'\ x * rbt\_set\_assn\ S\ p)\ (rbt\_set\_insert\ x\ p)$$
$$(Id\ x'\ x)\ rbt\_set\_assn\ (mop\_set\_insert\ t\ x'\ S)$$

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

$$|S| \leq |ys| \wedge |xs| + |ys| \leq |as|$$
$$\longrightarrow rbt\_insert\_t\ (|S| + 1) \leq (\lambda\_.\ rbt\_insert\_t\ (|as| + 1))\ S$$

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

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

5.2 Kruskal

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

The proof development follows this general structure: first we define the abstract algorithm for minimum weight basis in matroids (c.f. Figure 1a) and verify it. Then we instantiate it with the cycle matroid for forests in undirected graphs and refine the algorithm with the usage of equivalence classes. Figure 1b shows the last-but-one stage in the step-wise refinement process. In a last step we fix the vertices to be natural numbers and the domain of the disjoint-set data structure to be the set from `{0,...,M}`, with M being the maximal vertex in the graph. After that, we use the implementation of the union-find data structure from the TIICF to synthesize a concrete algorithm with the Sepref tool.

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

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

When developing this case study, we learned that the correctness arguments can be plainly reused, and that adding the proofs of the run-time claims does not interfere, as they only evoke additional verification conditions and leave the ones concerned with functional correctness unchanged. However, it is necessary to add more assertions in the algorithms that speak about the sizes of the data structures used. This reasoning is mostly done on the abstract level, but the information has to be passed to the concrete algorithm via assertions. In the Sepref translation phase, this information is needed to discharge the preconditions of the *hnr* predicates, which demand that enough time has been reserved to execute the step.

5.3 Edmonds–Karp: Reuse

Before starting this project we had the following working hypothesis:

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

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

The Edmonds–Karp algorithm repeatedly tries to increase the flow by searching for an augmenting path in the residual graph and terminates successfully if no such path exists. The search is conducted by a breadth-first search (BFS) on the residual graph. The structure of the development follows the original proof [13, 14]; we only give an abstract overview here:

First, an abstract BFS is defined and verified to return the shortest path from some start node to some end node. The algorithm is parametrized on some graph $G{=}(V{,}E)$ and some procedure that provides the successors of a node in that graph. The run-time of the BFS consequently depends on $|V|$ and $|E|$ as well as the run-time of the successor procedure and the allotted run-times for the data-structure operations used.

Second, an abstract Edmonds–Karp algorithm is defined assuming a procedure to find the shortest path in a graph. For that algorithm functional correctness is proven as well as the correct run-time bound depending on the underlying network, the run-time of the shortest-path algorithm and the run-times of the operations that maintain the residual graph.

Finally, by implementing the operations on the residual graph, in particular its successor function, the abstract algorithms can be interpreted and we obtain a concrete algorithm in NREST together with a refinement theorem and a compound run-time function. For that algorithm we synthesize a program in timed Imperative/HOL together with a correctness theorem and a run-time bound in $O(V * E * (E + V))$. Residual graphs are represented by matrices, for which we provide an array implementation in the TIICF, with linear-time initialization, and constant-time update and lookup operations.

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

6 Conclusion

6.1 Related Work

Lammich pioneered the Sepref tool [11] and it has been used to verify several interesting algorithms and software projects [13, 6, 16]. It was recently adapted to synthesize programs in LLVM [12] instead of Imperative/HOL. Coming up with a generic Sepref tool that is parametrized in the target and source language, as well as extending the LLVM semantics to run-time are interesting future projects.

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

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

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

6.2 Limitations and Future Work

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

In this paper we only study upper bounds of run-time of algorithms. This should be relaxed in two ways: First, consider other quantities, e.g. stack usage, or energy usage. Second, not only upper bounds can be reasoned about, also lower bounds are feasible. A refinement relation on lower bounds seems to be straightforward. Also combining this in a pair of *enat*s and keeping track of lower as well as upper bounds seems to be feasible.

We already mentioned, that Lammich's LLVM semantics could be extended to counting the number of operations. Obviously, it is future work to extend the collection of efficient data structures and reusable algorithms, as well as lowering the barriers to verify run-time arguments by providing more automation.

6.3 Conclusion

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

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

References

1. Robert Atkey. Amortised Resource Analysis with Separation Logic. In *ESOP*, volume 6012, pages 85–103. Springer, 2010.
2. Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkok, and John Matthews. Imperative functional programming with Isabelle/HOL. *Lecture Notes in Computer Science*, 5170:134–149, 2008.
3. Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 418–430, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034828.
4. Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, pages 1–35, 2017.
5. Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *International Conference on Computer Aided Verification*, pages 463–478. Springer, 2013.
6. Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 158–171. ACM, 2018.
7. Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*, 2018.
8. Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *International Conference on Interactive Theorem Proving*. Springer, 2019. URL: http://gallium.inria.fr/~agueneau/publis/gueneau-jourdan-charguer aud-pottier-2019.pdf.
9. Maximilian P.L. Haslbeck, Peter Lammich, and Julian Biendarra. Kruskal's Algorithm for Minimum Spanning Forest. *Archive of Formal Proofs*, February 2019. , Formal proof development. URL: http://isa-afp.org/entries/Kruskal.html.
10. Peter Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2014.
11. Peter Lammich. Refinement to Imperative/HOL. In *International Conference on Interactive Theorem Proving*, pages 253–269. Springer, 2015.
12. Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In *International Conference on Interactive Theorem Proving*. Springer, 2019.
13. Peter Lammich and S Reza Sefidgar. Formalizing the edmonds-karp algorithm. In *International Conference on Interactive Theorem Proving*, pages 219–234. Springer, 2016.
14. Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp Algorithm. *Archive of Formal Proofs*, August 2016. , Formal proof development. URL: http://isa-afp.org/entries/EdmondsKarp_Maxflow.html.
15. Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In *International Conference on Interactive Theorem Proving*, pages 166–182. Springer, 2012.
16. Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 61–78. Springer, 2018.
17. Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. In *International Joint Conference on Automated Reasoning*, pages 532–548. Springer, 2018.