For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM

MAXIMILIAN P. L. HASLBECK, Technische Universität München, Germany
PETER LAMMICH, University of Twente, Netherlands

We present a framework to verify both, functional correctness and (amortized) worst-case complexity of practically efficient algorithms. We implemented a stepwise refinement approach, using the novel concept of *resource currencies* to naturally structure the resource analysis along the refinement chain, and allow a fine-grained analysis of operation counts. Our framework targets the LLVM intermediate representation. We extend its semantics from earlier work with a cost model. As case studies, we verify the amortized constant time push operation on dynamic arrays and the $O(n \log n)$ introsort algorithm, and refine them down to efficient LLVM implementations. Our sorting algorithm performs on par with the state-of-the-art implementation found in the GNU C++ Library, and provably satisfies the complexity required by the C++ standard.

CCS Concepts: • **Theory of computation → Logic and verification**; **Separation logic**; **Program semantics**; **Program verification**.

Additional Key Words and Phrases: Algorithm Analysis, Program Verification, Refinement, Isabelle/HOL

ACM Reference Format:
Maximilian P. L. Haslbeck and Peter Lammich. 2021. For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM. *J. ACM* 37, 4, Article 111 (August 2021), 35 pages. https://doi.org/AA.BBBB/CCCCCCC.DDDDDDD

1 INTRODUCTION

In general, not only correctness, but also the complexity of algorithms is important. While it is obvious that the performance *observed* during experiments is essential to solve practical problems efficiently, also the *theoretical* worst-case complexity of algorithms is crucial: a good worst-case complexity avoids timing regressions when hitting worst-case input, and, even more important, prevents denial of service attacks that intentionally produce worst-case scenarios to overload critical computing infrastructure.

For example, the C++ standard requires implementations of *std::sort* to have worst-case complexity $O(n \log n)$ [8]. Note that this rules out quicksort [16], which is very fast in practice, but has quadratic worst-case complexity. Nevertheless, some standard libraries, most prominently LLVM's *libc++* [27], still use sorting algorithms with quadratic worst-case complexity.[1]

A practically efficient sorting algorithm with $O(n \log n)$ worst-case complexity is Musser's introsort [30]. It combines quicksort with the $O(n \log n)$ heapsort algorithm, which is used as fallback when the quicksort recursion depth exceeds a certain threshold. It allows to implement

[1] See, e.g., https://bugs.llvm.org/show_bug.cgi?id=20837.

Authors' addresses: Maximilian P. L. Haslbeck, haslbema@in.tum.de, Technische Universität München, Munich, Germany; Peter Lammich, University of Twente, Enschede, Netherlands, p.lammich@utwente.nl.

0004-5411/2021/8-ART111 $15.00
https://doi.org/AA.BBBB/CCCCCCC.DDDDDDD

standard-compliant, practically efficient sorting algorithms. Introsort is implemented by, e.g., the GNU C++ Library (*libstdc++*) [10].

In this paper, we present techniques to formally verify both, correctness and worst-case complexity of practically efficient implementations. Our approach seamlessly works for both standard and amortized analysis. We build on two previous lines of research by the authors.

On the one hand, we have the Isabelle Refinement Framework [26], which allows for a modular top-down verification approach. It utilizes stepwise refinement to separate the different aspects of an efficient implementation, such as algorithmic idea and low-level optimizations. It provides a nondeterminism monad to formalize programs and refinements, and the Sepref tool to automate canonical data refinement steps. Its recent LLVM back end [22] allows to verify algorithms with competitive performance compared to (unverified) highly optimized C/C++ implementations. The Refinement Framework has been used to verify the functional correctness of an implementation of introsort that performs on par with *libstdc++*'s implementation [24].

On the other hand, we already have extended the Refinement Framework to reason about complexity [14]. However, the cost model used there limits the natural structuring of the cost analysis in refinement proofs. Moreover, it only supports the Imperative HOL back end [23], which generates functional code that is inherently less efficient than imperative code, e.g., in C/C++.

This paper extends our conference paper [15] by adding amortized analysis and a case study on dynamic arrays, complexity analysis of string sorting, and more in-depth explanations of the design choices of our framework. We also make the paper more self-contained by including material from [14]. Our main contributions are.

- We present a generalized nondeterminism monad with resource cost, apply it to resource functions to model fine-grained currencies (Section 2), and show how they can be used to naturally structure refinement.
- We extend the LLVM back end [22] with a cost model, and amend its basic reasoning infrastructure (Section 3).
- We extend the Sepref tool (Section 4) to synthesize executable imperative code in LLVM, together with a proof of correctness and complexity.
- We show how to integrate the analysis of amortized data structures with our refinement approach (Section 5).
- We extend the verification of introsort to also show a worst-case complexity of $O(n \log n)$, thus meeting the C++11 *stdlib* specification [8] (Section 6). Our methodology also works for sorting data (e. g. strings) with a comparison operation that does not have constant running time. The performance of our implementation is still on par with *libstdc++*. We believe that this is the first time that both, correctness and complexity of a sorting algorithm have been formally verified down to a competitive implementation.

Our formalization is available at https://www21.in.tum.de/~haslbema/llvm-time.

2 SPECIFICATION OF ALGORITHMS WITH RESOURCES

We use the formalism of monads [35] to elegantly specify programs with resource usage. We first describe a framework that works for a very generic notion of *resource*, and then instantiate it with *resource functions*, which model resources of different *currencies*. We then describe a refinement calculus and show how currencies can be used to structure stepwise refinement proofs. Finally, we report on automation and discuss alternatives to our modelling of programs with resources.

2.1 Nondeterministic Computations With Resources

Let us examine the features we require for our computation model.

First, we want to specify programs by their desired properties, without having to fix a concrete implementation. In general, those programs have more than one correct result for the same input. Consider, e.g., sorting a list of pairs of numbers by the first element. For the input $[(1,2),(2,2),(1,3)]$, both $[(1,2),(1,3),(2,2)]$ and $[(1,3),(1,2),(2,2)]$ are valid results. Formally, this is modelled as a *set* of possible results. When we later fix an implementation, the set of possible results may shrink. For example, the (stable) insertion sort algorithm always returns the list $[(1,2),(1,3),(2,2)]$. We say that insertion sort *refines* our specification of sorting.

Second, we want to define recursion by a standard fixed-point construction over a flat lattice. The bottom of this lattice must be a dedicated element, which we call `fail`. It represents a computation that may not terminate.

Finally, we want to model the resources required by a computation. For nondeterministic programs, these may vary depending on the nondeterministic choices made during the computation. As we model computations by their possible results, rather than by the exact path in the program that leads to the result, we also associate resource cost with possible results. When more than one computation path leads to the same result, we take the supremum of the used resources. The notion of refinement is now extended to a subset of results that are computed using less resources.

We now formalize the above intuition: the type

$$(\alpha, \gamma)\ \mathit{NREST} = \texttt{fail} \mid \texttt{res}\ (\alpha \to \gamma\ \mathit{option})$$

models a nondeterministic computation with results of type α and resources of type γ.[2] That is, a computation is either `fail`, or `res` M, where M is a partial function from possible results to resources.

We define `spec` $\Phi\ T$ as a computation of any result r that satisfies $\Phi\ r$ using $T\ r$ resources: `spec` $\Phi\ T$ = `res` $(\lambda r.$ `if` $\Phi\ r$ `then` $\mathit{Some}\ (T\ r)$ `else` $\mathit{None})$. By abuse of notation, we write `spec` $x\ T$ for `spec` $(\lambda r.\ r = x)\ (\lambda\_.\ T)$.

Based on an ordering on the resources γ, we define the *refinement ordering* on NREST, by first lifting the ordering to *option* with *None* as the bottom element, then pointwise to functions and finally to (α, γ) *NREST*, setting `fail` as the top element. This matches the intuition of refinement: $m \le m'$ reads as *m refines m′*, i.e., m has less possible results than m', computed with less resources.

We require the resources γ to have a complete lattice structure, such that we can form suprema over the (possibly infinitely many) paths that lead to the same result. Moreover, when sequentially composing computations, we need to add up the resources. This naturally leads to a monoid structure $(\gamma, 0, +)$, where 0, intuitively, stands for no resources.

We call such types γ *resource types*, if they have a complete lattice and monoid structure. Note that, in an earlier iteration of this work [14], the resource type was fixed to extended natural numbers ($\mathit{enat} = \mathbb{N} \cup \{\infty\}$), measuring the resource consumption with a single number. Also note that (α, unit) *NREST* is isomorphic to our original nondeterministic result monad without resources [26].

If γ is a resource type, so is $\eta \to \gamma$. Intuitively, such resources consist of coins of different *resource currencies* η, the amount of coins being measured by γ.

Example 2.1. In the following we use the resource type $\mathit{ecost} = \mathit{string} \to \mathit{enat}$, i.e., we have currencies described by a string, whose amount is measured by extended natural numbers, where ∞ models arbitrary resource usage. Note that, while the resource type $\mathit{string} \to \mathit{enat}$ guides intuition, most of our theory works for general resource types of the form $\eta \to \gamma$ or even just γ.

We define the function $\$_s\ n$ to be the resource function that uses $n :: \mathit{enat}$ coins of the currency $s :: \mathit{string}$, and write $\$_s$ as shortcut for $\$_s\ 1$.

A program that sorts a list in $O(n^2)$ can be specified by:

[2]The name NREST abbreviates **N**ondeterministic **RES**ult with **T**ime, and has been inherited from our earlier formalizations.

$sort_{spec}\ xs = \texttt{spec}\ (\lambda xs'.\ sorted\ xs' \wedge mset\ xs' = mset\ xs)\ (\lambda\_.\ \$_q\ |xs|^2 + \$_c)$

That is, a list xs can result in any sorted list xs' with the same elements, and the computation takes (at most) quadratically many q coins in the list length, and one c coin, independently of the list length. Intuitively, the q and c coins represent the constant factors of an algorithm that implements that specification and are later elaborated by exchanging them into several coins of more fine-grained currencies, corresponding to the concrete operations in the algorithm, e.g., comparisons and memory accesses. *Abstract currencies* like q and c only *"have value"* if they can be exchanged to meaningful other currencies, and finally pay for the resource costs of a concrete implementation.

2.2 Atomic Operations and Control Flow

In order to conveniently model actual computations, we define some combinators. The `elapse` $m\ t$ combinator adds the (constant) resources t to all results of m:

$\texttt{elapse} :: (\alpha, \gamma)\ NREST \to \gamma \to (\alpha, \gamma)\ NREST$
$\texttt{elapse}\ \texttt{fail}\ t = \texttt{fail}$
$\texttt{elapse}\ (\texttt{res}\ M)\ t = \texttt{res}\ (\lambda x.\ \texttt{case}\ M\ x\ \texttt{of}\ None \Rightarrow None \mid Some\ t' \Rightarrow Some\ (t + t'))$

The program[3] `return` x computes the single result x without using any resources:

$\texttt{return} :: \alpha \to (\alpha, \gamma)\ NREST$
$\texttt{return}\ x = \texttt{res}\ [\ x \mapsto 0\ ]$

The combinator `bind` $m\ f$ models the sequential composition of computations m and f, where f may depend on the result of m:

$\texttt{bind} :: (\alpha, \gamma)\ NREST \to (\alpha \to (\beta, \gamma)\ NREST) \to (\beta, \gamma)\ NREST$
$\texttt{bind}\ \texttt{fail}\ f = \texttt{fail}$
$\texttt{bind}\ (\texttt{res}\ M)\ f = Sup\ \{\ \texttt{elapse}\ (f\ x)\ t\ |x\ t.\ M\ x = Some\ t\ \}$

If the first computation m fails, then also the sequential composition fails. Otherwise, we consider all possible results x with resources t of m, invoke $f\ x$, and add the cost t for computing x to the results of $f\ x$. The supremum aggregates the cases where f yields the same result, via different intermediate results of m, and also makes the whole expression fail if one of the $f\ x$ fails.

To improve readability of programs, we write $x \leftarrow m; f\ x$ for `bind` $m\ (\lambda x.\ f\ x)$ and, $m_1; m_2$ for `bind` $m_1\ (\lambda\_.\ m_2)$.

Example 2.2. We now illustrate an effect that stems from our decision to aggregate the resource usage of different computation paths that lead to the same result. Consider the program

$\texttt{res}\ (\lambda n{::}nat.\ Some\ (\$_c\ n)); \texttt{return}\ 0$

It first chooses an arbitrary natural number n consuming n coins of currency c, and then returns the result 0. That is, there are arbitrarily many paths that lead to the result 0, consuming arbitrarily many c coins. The supremum of this is ∞, such that the above program is equal to `elapse` (`return` 0) ($\$_c\ \infty$). Note that none of the computation paths actually attains the aggregated resource usage. We will come back to this in Section 4.5.

[3]Note that our shallow embedding makes no formal distinction between syntax and semantics. Nevertheless, we refer to an entity of type *NREST*, as *program* to emphasize the syntactic aspect, and as *computation* to emphasize the semantic aspect.

Finally, we use Isabelle/HOL's *if-then-else* and define a recursion combinator `rec` via a fixed-point construction [19], to get a complete set of basic combinators. As these combinators also incur cost in the target LLVM, we define resource aware variants. Furthermore we also derive a while combinator:

$\texttt{if}_c\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 = \texttt{elapse}\ (r \leftarrow b;\ \texttt{if}\ r\ \texttt{then}\ c_1\ \texttt{else}\ c_2)\ \$_{if}$
$\texttt{rec}_c\ F\ x = \texttt{elapse}\ (\texttt{rec}\ (\lambda D\ x.\ F\ (\lambda x.\ \texttt{elapse}\ (D\ x)\ \$_{call})\ x)\ x)\ \$_{call}$
$\texttt{while}_c\ b\ f\ s = \texttt{rec}_c\ (\lambda D\ s.\ \texttt{if}_c\ b\ s\ \texttt{then}\ s \leftarrow f\ s;\ D\ s\ \texttt{else return}\ s)\ s$

Here, the guard of $\texttt{if}_c$ is a computation itself, and we consume an additional *if* coin to account for the conditional branching in the target model. Similarly, every recursive call consumes an additional *call* coin.

Assertions fail if their condition is not met, and return *unit* otherwise:

$\texttt{assert}\ P = \texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else fail}$

They are used to express preconditions of a program. A Hoare triple for program m, with precondition P, postcondition Q and resource usage t is written as a refinement condition:

$m \le \texttt{assert}\ P;\ \texttt{spec}\ Q\ (\lambda\_.\ t)$

Example 2.3. Comparison of two list elements at a cost of t can be specified by:

$idxs\_cmp_{spec}\ xs\ i\ j\ (t) = \texttt{assert}\ (i < |xs| \wedge j < |xs|);\ \texttt{spec}\ (xs!i < xs!j)\ (\lambda\_.\ t)$

Here, the term *xs!i* is the *i*th element of list *xs*. Instead of fixing the cost for specifications, we pass them as parameter *t*. This allows us to refine different instances of abstract data types (here lists) by different concrete data structures with different costs. To make bigger programs more readable, we note the cost parameter in parenthesis at the end of the line, as, e.g., in Example 2.6.

Example 2.4. As another running example we consider the amortized constant time push operation of dynamic arrays. Abstractly, we specify appending an element at the end of a list.

$list\_push_{spec}\ xs\ x\ (t) = \texttt{spec}\ [xs \cdot [x] \mapsto t]$

Here, the term $xs \cdot ys$ denotes appending of two lists and we leave the amount of consumed resource t as a parameter. This specification has no precondition.

2.3 Refinement on NREST

We have used the refinement ordering to express Hoare triples. Two other applications of refinement are data refinement and currency refinement.

2.3.1 Data Refinement. A typical use-case of refinement is to implement an *abstract* data type by a *concrete* data type. For example, we could implement (finite) sets of numbers by sorted distinct lists. We define a *refinement relation* R between sorted distinct lists and sets. A concrete computation $m_\dagger$ that yields lists then refines an abstract computation m that yields sets, if every possible concrete result is related to a possible abstract result. Formally, $m_\dagger \le \Downarrow_D R\ m$, where the operator $\Downarrow_D$ is defined, for arguments R and m, by the following two rules.

$\Downarrow_D R\ (\texttt{res}\ M) = \texttt{res}\ (\lambda c.\ Sup\ \{M\ a \mid a.\ (c, a) \in R\}) \qquad \Downarrow_D R\ \texttt{fail} = \texttt{fail}$

Again, we use the supremum to aggregate the costs of all abstract results that are related to a concrete result. As in Example 2.2, this leads to the possibility that the supremum cost is not attained, which we discuss in Section 4.5.

Example 2.5. Reconsider the example of the dynamic array. We model dynamic arrays (*da*) first abstractly by *dynamic lists* (*dl*). They consist of a carrier list *cs* and two numbers *l* and *c* representing the *length* and the *capacity* of the dynamic list. A list *as* is refined by a dynamic list (*cs, l, c*), if the first *l* elements of *cs* form the list *as*. Furthermore, in a valid dynamic list the length is at most the capacity and the capacity is the length of the carrier list. Formally:

$$((cs,\ l,\ c),\ as) \in R^{list}_{dynlist} \longleftrightarrow take\ l\ cs = as \wedge l \leq c \wedge c = |cs|$$

Using this representation, we can now specify a push operation on dynamic lists. A push of an element x to a dynamic list (*cs, l, c*) will result in a valid dynamic list that contains the same elements as before and adds the element x at the end. As the dynamic list may have reached its capacity, it may be necessary to increase the capacity. We can state the intuition in the following NREST program:

$$dl\_push_{spec}\ (cs,\ l,\ c)\ x\ (t) = \mathtt{spec}\ (\lambda(cs',\ l',\ c').\ take\ l\ cs' = take\ l\ cs \wedge cs'\ !\ l = x \\ \wedge\ l' \leq c' \wedge c' = |cs'| \wedge l' = l + 1 \wedge c' \geq c)\ (\lambda\_.\ t)$$

Here, we first only specify the functional correctness, and leave the cost t as a parameter. We already fix that the program has constant cost, independent from the result and the input. The specification requires that the resulting dynamic list contains all the elements as before and adds x at the end. It is not specified whether or how much the carrier list has to increase.

We can now show that the push operation on dynamic lists refines the $list\_push_{spec}$ operation on lists:

$$[\![((cs,\ l,\ c),\ as) \in\ R^{list}_{dynlist},\ (x,\ x') \in Id]\!] \\ \Longrightarrow\ dl\_push_{spec}\ (cs,\ l,\ c)\ x\ (t) \leq \Downarrow_D\ R^{list}_{dynlist}\ (list\_push_{spec}\ as\ x'\ (t))$$

2.3.2 Currency Refinement. Consider we want to refine Example 2.3 into a program that first accesses the elements and then compares them.

Example 2.6. We refine $idxs\_cmp_{spec}$ ($\$_{idxs\_cmp}$) from Example 2.3 as follows:

$idxs\_cmp\ xs\ i\ j =$
$\quad \mathtt{assert}\ (i{<}|xs| \wedge j{<}|xs|);$
$\quad xsi \leftarrow list\_get_{spec}\ xs\ i;$ ($\$_{lookup}$)
$\quad xsj \leftarrow list\_get_{spec}\ xs\ j;$ ($\$_{lookup}$)
$\quad \mathtt{return}\ (xsi < xsj)$ ($\$_{less}$)

Where $list\_get_{spec}\ xs\ i\ (t) = \mathtt{assert}\ (i < |xs|);\ \mathtt{spec}\ (xs!i)\ (\lambda\_.\ t)$ and $\mathtt{return}\ x\ (t)$ returns the result x incurring cost t.

Note that *idxs_cmp* and $idxs\_cmp_{spec}$ use different, incompatible currency systems. To compare them, we need to exchange coins: one *idxs_cmp* coin will be traded for two *lookup* coins and one *less* coin.

To make that happen we introduce the currency refinement $\Downarrow_C E\ m$. Here, the *exchange rate* $E :: \eta_a \to \eta_c \to \gamma$ specifies for each abstract currency $c_a :: \eta_a$ how many of the coins of the concrete currency $c_c :: \eta_c$ are needed. Note that, in general, one abstract coin may be exchanged into multiple coins of different currencies. For a resource type γ that provides a multiplication operation ($*$) we define the operator $\Downarrow_C$ with the following two rules:

$$\Downarrow_C E\ (\mathtt{res}\ M) = \mathtt{res}\ (\lambda\ r.\ \mathtt{case}\ M\ r\ \mathtt{of}\ None \Rightarrow None \mid \\ Some\ t \Rightarrow Some\ (\lambda c_c.\ \textstyle\sum_{c_a}\ t\ c_a * E\ c_a\ c_c))$$
$$\Downarrow_C E\ \mathtt{fail} = \mathtt{fail}$$

The refined computation has the same results as the original. To get the amount of a concrete coin c_c for some result r with resource function t, we sum, over all abstract coins c_a, the amount of abstract coins needed in the original computation ($t\ c_a$) weighted by the exchange rate ($E\ c_a\ c_c$).

The sum only makes sense, if there are finitely many abstract coins c_a with $t\ c_a * E\ c_a\ c_c \neq 0$. This can be ensured by restricting the resource functions t of the computation to use finitely many different coins, or by restricting the exchange rate E accordingly. The latter can be checked syntactically in practice.

Example 2.7. For refining $idxs\_cmp_{spec}$ we define an exchange rate that does the correct exchange for currency $idxs\_cmp$ and is zero everywhere else. Formally: $E_1 = 0(idxs\_cmp := \$_{lookup}\ 2 + \$_{less})$. Here, + and 0 are lifted to functions in a pointwise manner, and $f(\cdot := \cdot)$ denotes a function update. We can now prove:

$$idxs\_cmp\ xs\ i\ j \leq \Downarrow_C E_1\ (idxs\_cmp_{spec}\ xs\ i\ j\ (\$_{idxs\_cmp}))$$

2.4 Notation for Refinement

When considering data refinement, we will often see propositions of the form

$$[\![\ P\ x\ x';\ (x, x') \in R\ ]\!] \implies f\ x \leq \Downarrow_D S\ (f'\ x)$$

This states that f refines f' w. r. t. relation R for the arguments and relation S for the result, if the additional precondition P holds for the arguments. To write those propositions more conveniently, we use the following notation[4]:

$$(f, f') \in [\lambda x\ x'.\ P\ x\ x']\ R \to S = (\forall x\ x'.\ P\ x\ x' \wedge (x, x') \in R \implies f\ x \leq \Downarrow_D S\ (f'\ x'))$$

If the precondition is always true, we just write $(f, f') \in R \to S$. For the sake of readability, we will identify curried and uncurried functions and write $(f, f') \in R_1 \to \ldots \to R_n \to S$ for programs with n arguments that are refined by $R_1, \ldots, R_n$.

The above form of those propositions is called the *parametric form*. It brings to mind relational parametricity by Wadler [34].

Example 2.8. Using that notation, the refinement from Example 2.5 reads as follows:

$$(dl\_push_{spec}\ (t),\ list\_push_{spec}\ (t)) \in R^{list}_{dynlist} \to Id \to R^{list}_{dynlist}$$

That is, if the parameters are related by $R^{list}_{dynlist}$ and Id, then the result of dl_push_{spec} refines the result of $list\_push_{spec}$ w. r. t. relation $R^{list}_{dynlist}$.

2.5 Refinement Patterns

In practice, we encounter certain recurring patterns of refinement, which we describe in this section.

Refinement of Specifications. A common application is to show that a program m satisfies a specification `res` Q, formally $m \leq$ `res` Q. Such proofs are usually done by a verification condition generator (VCG), that decomposes the program m according to its syntactic structure.

In a traditional setting without resources, we would use a notion of weakest precondition ($wp\ m\ Q = m \leq$ `res` Q), and define rules that syntactically decompose goals of the form $wp\ m\ Q$. For example, for sequential composition we have the rule:

$$wp\ m\ (\lambda x.\ wp\ (f\ x)\ Q) \implies wp\ (x \leftarrow m; f\ x)\ Q$$

[4]This notation was first described in [21, §2.2].

In a setting with time[5], however, this approach does not work, as the specification Q is not a predicate but a *deadline* of type $\alpha \to \gamma$ *option* that assigns any result a maximum allowed time.

We solve that problem by generalizing the concept of weakest preconditions from the *qualitative* to the *quantitative* domain: instead of only asking *whether* a program m satisfies a specification $\texttt{res}\ Q$, we ask *how much* it satisfies the specification, i. e. what is the *latest* feasible time at which we can start m to still match the deadline Q. We denote this by $gwp\ m\ Q :: \gamma$ *option* (*generalized weakest precondition*). If the specification is not satisfied, we have $gwp\ m\ Q = None$. In particular, we have the following equalities: $m \le \texttt{res}\ Q \Leftrightarrow gwp\ m\ Q \neq None \Leftrightarrow Some\ 0 \le gwp\ m\ Q$. Our VCG now operates on goals of the form $Some\ t \le gwp\ m\ Q$, and the sequential composition rule reads:

$$Some\ t \le gwp\ m\ (\lambda x.\ gwp\ (f\ x)\ Q) \implies Some\ t \le gwp\ (x \leftarrow m; f\ x)\ Q$$

Formally, we define the generalized weakest precondition as follows:

$$gwp\ \texttt{fail}\ Q = None$$
$$gwp\ (\texttt{res}\ M)\ Q = Inf\ r.\ minus\ (Q\ r)\ (M\ r)$$

That is, if the program fails, no starting time is feasible, as expressed by *None*. Otherwise, we use the most conservative starting time over all possible results, expressed by the infimum (Inf). For a single result, the latest feasible starting time is expressed by the difference of the resources specified and actually used. The difference operator $minus :: \gamma\ option \to \gamma\ option \to \gamma\ option$ lifts the difference on resources[6] to option types. Note that, if the specification cannot be met due to a single result r, the difference is *None*, causing the infimum to be *None*. Formally, we distinguish the following cases:

- $minus\ (Some\ t')\ (Some\ t) = \texttt{if}\ t' \ge t\ \texttt{then}\ Some\ (t' - t)\ \texttt{else}\ None$: if the difference is not negative, we return it. Otherwise, the program consumes more resources than specified and does not meet the specification.
- $minus\ None\ (Some\ t) = None$: the result is not covered by the specification, hence the specification cannot be met.
- $minus\ \_\ None = Some\ \top$: the result is not produced by the program, thus it does not contribute to the latest feasible starting time. Accordingly, we return the top element $Some\ \top$.

It is straightforward to define gwp rules for our monad operations, and construct the desired syntax driven VCG. For details, we refer the reader to [14].

Lockstep Refinement. We often refine a compound program by refining some of its components. Let A and C be two structurally equal programs (i.e., they have the same structure of combinators $\texttt{if}_c$, $\texttt{rec}_c$, $\texttt{bind}$, etc.), and let A_i and C_i be the pairs of corresponding basic components, for $i \in \{0, \ldots, n\}$. Provided with refinement lemmas $(C_i, \lambda x.\ \Downarrow_C E\ (A_i\ x)) \in [\Phi_i]\ R_i \to S_i$ for each of those pairs,[7] an automatic procedure walks through the program and establishes a refinement $(C, \lambda x.\ \Downarrow_C E\ (A\ x)) \in [\Phi]\ R \to S$. This process generates verification conditions for ensuring the preconditions Φ_i, which can be discharged automatically or, if required, via interactive proof.

Note that, while the data refinements R_i can be different for each component i, the exchange rate E must be the same for all components. Currently, we align the exchange rates by manually deriving specialized versions of the component refinement lemmas. However, we believe that this can be automated in many practical cases, by collecting constraints on the exchange rate during

[5]To guide the intuition, we will use time as resource here.
[6]This requires γ to provide a difference operator, dual to its + operator. It is a straightforward generalization of the concept defined in [14]. We note that the resource types $unit$, $enat$, and $ecost$ provide a suitable difference operator.
[7]The refinement relations R_i and S_i relate the parameters and respectively the result of those components.

the lockstep refinement, which are solved afterwards to obtain a unified exchange rate. We leave the implementation of this idea to future work.

Separating Analysis of Resource Usage and Correctness. We can disregard resource usage and only focus on refinement of functional correctness, and then add resource usage analysis later. This is useful to separate the concerns of functional correctness and resource usage proof. We will describe a practical example in Section 6.5. Here, we only present an alternative way to prove the refinement from Example 2.6:

First, for functional correctness, we use the specification $idxs\_cmp_{spec}\ (\infty)$ and a program $idxs\_cmp_\infty$ similar to *idxs_cmp* but with all the costs replaced by ∞. Proving the refinement $idxs\_cmp_\infty\ xs\ i\ j \le idxs\_cmp_{spec}\ xs\ i\ j\ (\infty)$ only requires showing verification conditions that correspond to functional properties and termination, in particular those from assertions and annotated invariants in the concrete program. Proof obligations on resource usage, however, collapse into the trivial $t \le \infty$. For the same reason, we get $idxs\_cmp\ xs\ i\ j \le idxs\_cmp_\infty\ xs\ i\ j$, and, by transitivity:

$$idxs\_cmp\ xs\ i\ j \le idxs\_cmp_{spec}\ xs\ i\ j\ (\infty)$$

Next, we prove $idxs\_cmp\ xs\ i\ j \le_n \texttt{spec}\ (\lambda\_.True)\ (\$_{lookup}\ 2 + \$_{less})$. Here, the refinement relation $m \le_n m' = (m \neq \texttt{fail} \implies m \le m')$ *assumes* that the concrete program does *not* fail. This has the effect that, during the refinement proof, assertions and annotated invariants in the concrete program can be assumed to hold, and we can focus on the resource usage proof.

Finally, the two refinements can be combined to obtain

$$idxs\_cmp\ xs\ i\ j \le idxs\_cmp_{spec}\ xs\ i\ j\ (\$_{lookup}\ 2 + \$_{less})$$

2.6 Alternatives to NREST

In the beginning of this section we stated our motivations and design goals for NREST. To model nondeterminism and resources, we used partial functions that map results to resource elements. To motivate this design, we discuss some seemingly obvious alternatives.

A result set and a resource. An alternative would be to define an NREST program being a set of results together with a single resource element for all possible results:

$$(\alpha, \gamma)\ NREST_1 = \texttt{fail} \mid \texttt{res}\ (\alpha\ set \times \gamma)$$

However, this modelling is too coarse: consider a program that modifies a set of natural numbers by repeating the following step until the set is empty: pick and remove a number n from the set, then consume n resources.

Say we start with a set $\{1, 2\}$. Then, the result after the first step is $\texttt{res}\ (\{\ \{1\}, \{2\}\ \}, 2)$, as there are two possibilities which element was removed from the set, and the upper bound of both outcomes is 2. After the second step the result must be $\texttt{res}\ (\{\ \emptyset\ \}, 4)$, as in both cases the remaining element is removed, but again the upper bound on the running time of that second step is 2. This yields a total running time of 4, which is not tight.

In order to use nondeterminism effectively, we need a finer assignment of resources to results.

A set of pairs. Another alternative is to regard the resource usage just as part of the result. Thus, a set of results with resource usage would be modeled as $(\alpha \times \gamma)\ set$. Note that this is isomorphic to $\alpha \to \gamma\ set$, which suits our presentation better. So we define the following alternative to NREST:

$$(\alpha, \gamma)\ NREST_2 = \texttt{fail} \mid \texttt{res}\ (\alpha \to \gamma\ set)$$

On the one hand, this definition certainly allows to model the two stage process from above adequately. Depending on which number out of $\{1, 2\}$ was chosen we can specify a different resource consumption for the intermediate results, and in the end model a tight running time of 3.

On the other hand, the refinement relation cannot just be the natural subset relation, because we would like to have e.g. $\{(x, 3), (x, 4)\} \leq \{(x, 4)\}$, in order to allow refinement with programs with less resource consumption. Formally, we can use a *downward closure* $(\cdot^{\downarrow})$ to express refinement:

$$S^{\downarrow} = \{s \mid \exists s' \in S.\ s \leq s'\}$$
$$\texttt{res}\ M \leq \texttt{res}\ M' = \forall x.\ M\ x \subseteq (M'\ x)^{\downarrow}$$

That is, the computation `res` M refines `res` M' if for all results x in M the set of possible resource costs is bounded by some possible resource bound for x in M'.

In our initial design considerations for NREST we dropped that approach because it felt unnatural and the alternative to map results to single resource elements worked out more smoothly. In the following we present some results of a later effort to use the "set of pairs" approach.

First, we note that the refinement defined with the downward closure as above is not anti-symmetric, and thus yields no complete lattice structure. This problem, however, can be easily solved by identifying sets with the same downward closure. Technically, we use the quotient type $\gamma\ \mathit{dclosed} = \gamma\ \mathit{set}\ /\ (\lambda s_1\ s_2.\ {s_1}^{\downarrow} = {s_2}^{\downarrow})$, and define a new variant of NREST accordingly:

$$(\alpha, \gamma)\ \mathit{NREST}_3 = \texttt{fail} \mid \texttt{res}\ (\alpha \rightarrow \gamma\ \mathit{dclosed})$$

For this, we straightforwardly get the desired complete lattice structure on NREST_3. We even get a more elegant formalization, as the empty set $(\emptyset^{\downarrow})$ naturally models the case where no result is present, and the universal set $(\mathit{UNIV}^{\downarrow})$ is the greatest element. In our original NREST, we had to use partial functions to model absence of results, and add artificial greatest elements to the resource type (e.g., ∞ in *enat*).

For a resource type that provides a neutral element 0 and addition $+$ with a monoid structure, we further can define the monadic operators `return`, `bind` and `elapse` as expected. The lifting of $+$ to downward closed sets, as required for defining `bind`, is straightforward.

However, we got stuck when we tried to define generalized weakest preconditions (cf. Section 2.5) in NREST_3, more precisely, the underlying difference operator on resources. For example, consider the following scenario where resources have more than one extreme point: we assume resources with two currencies, expressed as pairs of amounts. Let $\{\ (2, 0), (0, 2)\ \}^{\downarrow}$ be the specified resources for some result and $\{(1, 0), (0, 1)\}^{\downarrow}$ the ones actually required by the program. In order to determine *gwp*, we would have to take the difference of these two downward closed sets. However, it is unclear to us how to define the difference in a sensible way.

In our actual NREST design, however, we aggregate the cost into one element. We would obtain $(2, 2)$ and $(1, 1)$ respectively, and the difference operator can easily be defined pointwise. We have to note that the overapproximation of $\{\ (2, 0), (0, 2)\ \}$ to $(2, 2)$ *does* cause a problem, which we will treat in Section 4.5.

In summary, our choice of modeling NREST by one resource element per possible result seems to be a sweet spot: it is fine enough to model nondeterminism effectively and coarse enough to define generalized weakest preconditions.

3 LLVM WITH COST SEMANTICS

The NREST-monad allows to specify programs with their resource usage in abstract currencies. Those currencies only have a meaning when they finally can be exchanged for the costs of concrete computations. In the following we present such a concrete computation model, namely a shallow embedding of the LLVM semantics into Isabelle/HOL. The embedding is an extension of our earlier

work [22] to also account for costs. In Section 4 we will then report on linking the LLVM back end with the NREST front end.

3.1 Basic Monad

At the basis of our LLVM formalization is a monad that provides the notions of non-termination, failure, state, and execution costs.

$\alpha\ mres = NTERM \mid FAIL \mid SUCC\ \alpha\ cost\ state$
$\alpha\ M = state \rightarrow \alpha\ mres$

Here, *cost* is a type for execution costs, which forms a monoid with operation + and neutral element 0, and *state* is an arbitrary type.[8]

The type $\alpha\ M$ describes a program that, when executed on a state, either does not terminate (*NTERM*), fails (*FAIL*), or returns a result of type α, its execution costs, and a new state (*SUCC*).

It is straightforward to define the monad operations `return` and `bind`, as well as a recursion combinator `rec` over *M*. Thanks to the shallow embedding, we can also use Isabelle HOL's *if-then-else* to get a complete set of basic operations. As an example, we show the definition of the `bind` operation, in the case that both arguments successfully compute a result:

Assume $m\ s = SUCC\ x\ c_1\ s_1$ and $f\ x\ s_1 = SUCC\ r\ c_2\ s_2$
then we have `bind` $m\ f\ s = SUCC\ r\ (c_1{+}c_2)\ s_2$

That is, the result x and state s_1 after the first operation m is passed into the second operation f, and the result and state after the `bind` is what emerges from f. The cost for the `bind` is the sum of the costs for both operations.

The basic monad operations do not cost anything. To account for execution costs, we define an explicit operation `consume` $c\ s = SUCC\ ()\ c\ s$.[9]

3.2 Shallowly Embedded LLVM Semantics

The formalization of the LLVM semantics is organized in layers. At the bottom, there is a memory model that stores deeply embedded values, and comes with basic operations for allocation/deallocation, loading, storing, and pointer manipulation. Also the basic arithmetic operations are defined on deeply embedded integers. These operations are phrased in the basic monad, but consume no costs. This way, we could take them unchanged from our original LLVM formalization without cost [22]. For example, the low-level load operation has the signature $raw\_load :: raw\_ptr \rightarrow val\ M$. Here, *raw_ptr* is the pointer type of our memory model, consisting of a block address and an offset, and *val* is our value type, which can be an integer, a pointer, or a pair of values.

On top of the basic layer, we define operations corresponding to the actual LLVM instructions. Here, we map from deeply to shallowly embedded values, and add the execution costs.

For example, the semantics of LLVM's load instruction is defined as follows:

$ll\_load :: \alpha\ ptr \rightarrow \alpha\ M$
$ll\_load\ p =$
  `consume` $\$_{load}$;
  $r \leftarrow raw\_load\ (the\_raw\_ptr\ p)$;
  $checked\_from\_val\ r$

[8]Note that this differs from the NREST monad in Section 2.1: it is deterministic, and provides a state. Because of determinism, we never need to form a supremum, and thus can base our cost model on natural numbers rather than enats. We leave a unification of the two monads to future work.
[9]For NREST, we defined a *higher-order* operation `elapse`, while we use the *first-order* operation `consume` here. This is for historical reasons. Note that `elapse` can be defined in terms of `consume`, and vice versa.

It consumes the cost[10] for the operation, and then forwards to the *raw_load* operation of the lower layer, where *the_raw_ptr* and *checked_from_val* convert between the shallow and deep embedding of values.

Like in the original formalization[11], an LLVM program is represented by a set of monomorphic constant definitions of the shape *def*, defined as follows:

def = *proc_name var** ≡ *block*
block = *var* ← *cmd; block* | `return` *var*
cmd = *ll_<opcode> arg** | *ll_call proc_name arg** | *llc_if arg block block*
| *llc_while block block*
arg = *var* | *number* | *null* | *init*

The code generator checks that the set of definitions is complete and adheres to the required shape. It then translates them into LLVM code, which merely amounts to pretty printing and translating the structured control flow by `if` and `while`[12] statements to the unstructured control flow of LLVM. A powerful preprocessor can convert a more general class of terms to the restricted shape required by the code generator. This conversion is done inside the logic, i.e., the processed program is proved to be equal to the original. Preprocessing steps include monomorphization of polymorphic constants, extraction of fixed-point combinators to recursive function definitions, and conversion of tuple constructors and destructors to LLVM's *insertvalue* and *extractvalue* instructions.

In summary, the layered architecture of our LLVM formalization allowed for a smooth integration of the cost aspect, reusing most of the existing formalization nearly unchanged. Note that we opted to integrate the cost aspect into the existing top layer, which converts between deep and shallow embedding. Alternatively, we could have added another layer on top of the shallow embedding. While the latter would have been the cleaner design, we opted for the former approach to avoid the boilerplate of adding a new layer. This was feasible as the original top layer was quite thin, such that adding another aspect there did not result in excessive complexity.

3.3 Cost Model

As a cost model for running time, we chose to count how often each instruction is executed. That is, we set *cost* = *string* → *nat*, where the string encodes the name of an instruction. It is straightforward to define *0* and *+* such that (*cost, 0, +*) forms a monoid. It is thus a valid cost model for our monad.

But how realistic is our cost model, counting LLVM instructions? During compilation, LLVM text will be transformed by LLVM's optimizer, and finally, the LLVM back end will translate LLVM instructions to machine instructions. Moreover, the actual running time of a machine program does not only depend on the number of executed instructions, but effects like pipeline flushes and cache misses also play an important role. Thus, without factoring in the details of the optimization passes and the target machine architecture, our cost model can, at best, be a rough approximation of the actual running time.

However, we can sensibly assume that a single instruction in the original LLVM text will result in at most a (small) constant number of machine instructions, and that each machine instruction has a constant worst-case execution time. Thus, the steps counted by our model linearly correlate

[10] See Section 3.3 for an explanation of our cost model.
[11] Actually, the only change to the original formalization is the introduction of the *ll_call* instruction, to make the costs of a function call visible.
[12] Primitive while loops are not strictly required, as they can always be replaced by tail recursion. Indeed, our code generator can be configured to not accept while loops, and our preprocessor can automatically convert while loops to tail-recursive functions. However, the efficiency of the generated code then relies on LLVM's optimization pass to detect the tail recursion and transform it to a loop again.

to an upper bound of the actual execution time, though the exact correlation depends on the actual program, optimizer passes, and target architecture. Hence, while our cost model cannot be used for precise statements about execution time, it can be used to prove worst-case complexity. That is, a program that we have proved efficient will be compiled to an efficient machine program. Moreover, we can hope that the constant factors in the proved complexity are related to the actual constant factors in the machine program, i.e., an LLVM program with small constant factors will compile to a machine program with small constant factors.

The above discussion justifies the following design choices: The *insertvalue* and *extractvalue* instructions, which are used to construct and destruct tuple values, have no associated costs. The main reason for this design is to enable transparent use of tupled values, e.g., to encode the state of a while loop. We expect LLVM to translate the members of the tuple to separate registers anyway, such that no real costs are associated with tupling/untupling.

We define the *malloc* instruction to take cost proportional to the number of allocated elements. Note that LLVM itself does not provide memory management, and our code generator forwards memory management instructions to the *libc* implementation of the target platform. We use the *calloc* function here, which is supposed to initialize the allocated memory with zeros. While the exact costs of that are implementation dependent, they certainly will depend on the size of the allocated block.

Charguéraud and Pottier [7, §2.7] discuss the adequacy of abstract cost models in a functional setting. In their classification, our abstraction would be on Level 2, as we count (almost) all kinds of operations on an intermediate language level.

3.4 Reasoning Setup

Once we have defined the semantics, we need to set up some basic reasoning infrastructure. The original Isabelle-LLVM already comes with a quite generic separation logic and verification condition generation framework. Here, we report on our extensions to resources using time credits.

Separation Logic with Time Credits. Our reasoning infrastructure is based on separation logic with time credits [1, 7, 13]. We follow the algebraic approach of Calcagno *et al.* [3], using an earlier extension [22] of Klein *et al.* [25].

A separation algebra on type α induces a *separation logic* on assertions that are predicates over α. To guide intuition, elements of α are called *heaps* here. We use the following separation logic operators: The assertion $\uparrow\Phi$ holds for an empty heap if Φ holds, $\square = \uparrow True$ describes the empty heap, and $\exists_A$ is the existential quantifier lifted to assertions. The *separating conjunction* $P \star Q$ describes a heap comprised from two disjoint parts, one described by P and the other described by Q, and entailment $P \vdash Q$ states that Q holds for every heap described by P.

Separation algebras naturally extend over product and function types, i.e., for separation algebras α, β, and any type γ, also $\alpha \times \beta$ and $\gamma \to \alpha$ are separation algebras, where the operations are lifted pointwise.

Note that *enat* forms a separation algebra, where elements, i.e. time credits, are always disjoint. Hence, also $ecost = string \to enat$, and $amemory \times ecost$ are separation algebras, where *amemory* is the separation algebra that we already used in [22] to describe the abstract memory of LLVM. Thus, $amemory \times ecost$ induces a separation logic with time credits that match our cost model. The *time credit assertion* $\$\, c = (\lambda a.\ a = (0, c))$ describes an empty memory (0) and precisely the time c. The primitive assertions on *amemory* are lifted analogously to describe no time credits.

Weakest Precondition and Hoare Triples. We start by defining a concrete state *cstate* that describes the memory content and the available resources:

$$cstate = memory \times ecost$$

where *memory* is the memory type from our original LLVM formalization. Based on this, we define the weakest precondition predicate:

$$wp :: \alpha\ M \to (\alpha \to cstate \to bool) \to cstate \to bool$$
$$wp\ m\ Q\ (s, cc) = (\exists r\ c\ s'.\ m\ s = SUCC\ r\ c\ s' \wedge c \le cc \wedge Q\ r\ (s', cc - c)).$$

Intuitively, the costs *cc* stored in the state is the *credit* available to the program. The weakest precondition holds if the program runs with real costs *c* that are within the available credit, and *Q* holds for the result *r*, the new memory *s′*, and the new credit, *cc*−*c*, which is the old credit reduced by the actually required costs. Note that actual costs have type $cost = string \to nat$, i.e., are always finite, while the credits have type $ecost = string \to enat$, i.e., there can be infinite credits. Setting the credit to be infinite for all instruction types yields the classical weakest precondition that requires termination, but enforces no time limit.

Our concrete state type, in particular the memory, does not form a separation algebra, as the natural memory model of LLVM has no notion of partial memories. Thus, we define an abstraction function that maps a concrete state to an abstract state *astate*, which forms a separation algebra:

$$astate = amemory \times ecost \qquad abs\ (m, c) = (abs_m\ m, c)$$

Again, *amemory* and abs_m are the abstract state and abstraction function from the original LLVM formalization. The costs already form a separation algebra, so we do not abstract them further.

With this, we can instantiate a generic VCG infrastructure: let *cstate* be the type of concrete states, $wp :: \alpha\ M \to (\alpha \to cstate \to bool) \to cstate \to bool$ be a weakest precondition predicate, and *astate* the type of abstract states, linked to concrete states via an abstraction function $abs :: cstate \to astate$. Further, assume that *wp* distributes over conjunctions, i.e.,

$$wp\ c\ Q_1\ s \wedge wp\ c\ Q_2\ s \implies wp\ c\ (\lambda r\ s'.\ Q_1\ r\ s' \wedge Q_2\ r\ s')\ s$$

Finally, let $\top\!\!\top$ be an *affine top* [5], i.e., an assertion with $\square \vdash \top\!\!\top$ and $\top\!\!\top \star \top\!\!\top = \top\!\!\top$, which captures resources that can be safely discarded. We define the *Hoare triple* $\{P\}\ c\ \{Q\}$ to hold iff:

$$\forall F\ s.\ (P \star F)\ (abs\ s) \implies wp\ c\ (\lambda r\ s'.\ (Q\ r \star \top\!\!\top \star F)\ (abs\ s'))\ s$$

Intuitively, $\{P\}\ c\ \{Q\}$ holds if, for all states that contain a part described by assertion *P*, command *c* terminates with result *r* and a state where that part is replaced by a part described by $Q\ r \star \top\!\!\top$, and the rest of the state has not changed. Here, $Q\ r$ is the postcondition of the Hoare triple, and $\top\!\!\top$ describes resources that may be left over and can be discarded.

In our case, we set $\top\!\!\top$ to describe the empty memory and any amount of time credits. This matches the intuition that a program must free all its memory, but may run faster than estimated, i.e., leave over some time credits. Note that our *wp* distributes over conjunctions.

The generic VCG infrastructure now provides us with a syntax driven VCG with a simple frame inference heuristics.

3.5 Primitive Setup

Once we have defined the basic reasoning infrastructure, we have to prove Hoare triples for the basic LLVM instructions and control flow combinators. As we have added the cost aspect only at the top level of our semantics, we can reuse most of the material from our original LLVM formalization without time. Technically, we instantiate our reasoning infrastructure with a weakest precondition predicate *wpn*, which only holds for programs that consume no costs. We define:

$$wpn\ m\ Q\ s = wp\ m\ (FST \circ Q)\ (s, 0)\ \textbf{where}\ FST\ P = \lambda(s, c).\ P\ s \wedge c{=}0$$

The resulting reasoning infrastructure is identical with the one of our original formalization, most of which could be reused. Only for the topmost level, i.e., for those functions that correspond to the functional semantics of the actual LLVM instructions, we lift the Hoare triples over *wpn* to Hoare triples over *wp*:

$$\{P\}\ c\ \{Q\}_{wpn} = \{FST\ P\}\ c\ \{FST \circ Q\}$$

Example 3.1. Recall the low-level *raw_load* and the high-level *ll_load* instruction from Section 3.2. The *raw_load* instruction consumes no costs, and our original LLVM formalization provides the following Hoare triple:

$$\{raw\_pto\ p\ x\}\ raw\_load\ p\ \{\lambda r.\ \uparrow(r = x) \star raw\_pto\ p\ x\}_{wpn}$$

This can be transferred to a Hoare triple over *wp*:

$$\{FST\ (raw\_pto\ p\ x)\}\ raw\_load\ p\ \{\lambda r.\ \uparrow(r = x) \star FST\ (raw\_pto\ p\ x)\}$$

which is then used to prove the Hoare triple for the program *ll_load*

$$\{pto\ p\ x \star \$\ \$_{load}\}\ ll\_load\ p\ \{\lambda r.\ \uparrow(r = x) \star pto\ p\ x\}$$

where $pto\ p\ x = FST\ (raw\_pto\ (the\_raw\_ptr\ p)\ (to\_val\ x))$.

Using the VCG and the Hoare triples for the LLVM instructions, we can now define and prove correct data structures and algorithms. While this works smoothly for simple data structures like arrays, it does not scale to more complex developments. In contrast, NREST *does* scale, but lacks support for the low-level pointer reasoning required for basic data structures. In Section 4, we show how to combine both approaches, with the LLVM level providing basic data structures and the NREST level using them as building blocks for larger algorithms.

3.6 Free for Free

Note that in our semantics, both memory allocation and memory deallocation consume costs of currencies *malloc* and *free* respectively. However, the automatic data refinement tool we are going to design (see Section 4.2) has to automatically insert destructors, which free memory. A destructor d that destroys an object described by assertion A is characterized in the following way:

$$destructor\ A\ d = (\forall a\ c.\ \{A\ a\ c\}\ d\ c\ \{\square\})$$

In particular, all costs required for destruction must already be contained in the assertion A. In practice, this means that we pay for the destruction of an object upon its allocation. Thus, we prove the following Hoare triples for allocation and deallocation:

$$\{\ \$\ (\$_{malloc}\ n + \$_{free}) \star \uparrow(n > 0)\}$$
$$ll\_malloc\ \alpha\ n$$
$$\{\lambda p.\ range\ \{0..<n\}\ (\lambda\_.\ init)\ p \star malloc\_tag\ n\ p\}$$

$$\{range\ \{0..<n\}\ blk \star malloc\_tag\ n\ p\}\ ll\_free\ p\ \{\square\}$$

Intuitively, to allocate a block of size n, one has to pay n units of *malloc* and *1* unit of *free*. To free a block, no explicit costs have to be paid.

Note that the *malloc_tag* assertion in the original formalization expresses ownership on the whole block and is a prerequisite for freeing a block. Thus, it was natural to add the required time credits for freeing to this assertion, when extending the original formalization with time:

$$malloc\_tag\ n\ p = FST\ (raw\_malloc\_tag\ n\ (the\_raw\_ptr\ p)) \star \$\ \$_{free}$$

where *raw_malloc_tag* is the ownership assertion from our low-level memory model.

Note how amortization arguments like the above are seamlessly supported by separation logic with time credits [1]. Later in this paper (Section 5) we also show how to combine amortization with refinement.

In practice, the *malloc_tag* assertion is usually hidden in the assertion for a data structure, and thus not directly visible to the user.

3.7 Modelling Data Structures

An imperative data structure is described by a *refinement assertion* that relates it to a functional model. The refinement assertion usually contains the addresses and block ownership (*malloc_tag*) for all memory used to represent the data structure. For each operation, a Hoare triple is proved that relates the concrete operation on the heap to the corresponding abstract operation on the functional model.

For example, the assertion $array_A\ xs\ p$ relates the array pointed to by p to the list xs of its elements:

$$array_A\ xs\ p = range\ \{0..<|xs|\}\ (\lambda i.\ xs\ !\ i)\ p \star malloc\_tag\ |xs|\ p$$

The following Hoare triples relate the standard array operations to the corresponding operations on lists:

$$\{\$\ (\$_{malloc}\ n + \$_{free}) \star \uparrow(n > 0)\}\ array\_new\ \alpha\ n\ \{\lambda p.\ array_A\ (replicate\ n\ init)\ p\}$$
$$\{array_A\ xs\ p\}\ array\_delete\ p\ \{\square\}$$
$$\{array_A\ xs\ p \star\ \$\ (\$_{ofs\_ptr} + \$_{load}) \star \uparrow(i < |xs|)\}\ array\_nth\ p\ i\ \{\lambda r.\ array_A\ xs\ p \star \uparrow(r = xs\ !\ i)\}$$
$$\{array_A\ xs\ p \star\ \$\ (\$_{ofs\_ptr} + \$_{store}) \star \uparrow(i < |xs|)\}\ array\_upd\ p\ i\ x\ \{\lambda r.\ array_A\ (xs[i:=x])\ r\}$$

Users of the array data structure only need to use this interface, and never have to look into the details of the implementations or the refinement assertion.

Note that, as described in Section 3.6, we pay the cost for destruction already upon construction. For a simple array, the destructor only invokes *ll_free*, whose costs are already contained in *malloc_tag*. More complicated data structures, however, may require additional costs for destruction (e.g. to traverse a list of allocated arrays). These can also be hidden in the refinement assertion.

4 AUTOMATIC REFINEMENT

In this section we describe a tool to synthesize a concrete program in the LLVM-monad from an abstract algorithm in the NREST-monad. It can automatically refine abstract functional data structures to imperative heap-based ones. We will describe the synthesis predicate *hnr* that connects the two monads, the synthesis tool, and a way to extract Hoare triples from *hnr* predicates. Finally, we will discuss an effect that prevents combining *hnr* with data refinements in the NREST-monad in the general case.

4.1 Heap nondeterminism refinement

The *heap nondeterminism refinement* predicate $hnr\ \Gamma\ m_\dagger\ \Gamma'\ A\ m$ intuitively expresses that the concrete program $m_\dagger$ computes a concrete result that relates, via the refinement assertion A, to a result in the abstract program m, using at most the resources specified by m for that result. A refinement assertion describes how an abstract variable is refined by a concrete value on the heap. It can also contain time credits. The assertions Γ and Γ' constitute the heaps before and after the computation and typically are a separating conjunction of refinement assertions for the respective parameters of $m_\dagger$ and m. Formally, we define:

$$
\begin{array}{l}
\mathit{hnr}\,\Gamma\ m_\dagger\ \Gamma'\,A\ m = m \neq \texttt{fail} \implies \\
\quad (\forall F\ s\ c.\ (\Gamma \star F)\ (\mathit{abs}_m\ s,\ c) \implies \\
\qquad (\exists r_a\ c_a.\ \texttt{elapse}\ (\texttt{return}\ r_a)\ c_a \leq m \\
\qquad\quad \wedge\ \mathit{wp}\ m_\dagger\ (\lambda r\,(s',\ c').\ (\Gamma' \star A\ r\ r_a \star F \star \top)\ (\mathit{abs}_m\ s',\ c'))\ (s,\ c + c_a)))
\end{array}
$$

The predicate holds if either the abstract program fails or if, for all heaps and resources $(s,\ c)$ that satisfy the pre-assertion Γ with some frame F, there exists an abstract result and cost $(r_a,\ c_a)$ that refine m, and $m_\dagger$ terminates with concrete result r in a state s' where Γ' with the frame holds, and r relates to the abstract result via assertion A. The execution costs of $m_\dagger$ and the time credits c' required by the post-assertion Γ' are paid for by the specified cost c_a and the time credits c described by the pre-assertion Γ. Thus, the real costs are paid by a combination of the advertised costs in the abstract program and the potential difference of Γ' and Γ, allowing to seamlessly model amortized computation costs.

The affine top $\top$ allows the program to throw away portions of the heap. Note that our $\top$ can only discard time credits. Memory must be explicitly freed by the concrete program $m_\dagger$.

Also note that *hnr* is not tied to the LLVM semantics specifically. It actually is a general pattern for combining the NREST-monad with any other program semantics that provides a weakest precondition and a separation algebra for data and resources.

4.2 The Sepref Tool

The Sepref tool [20, 22] automatically synthesizes a concrete program in the LLVM-monad from an abstract algorithm in the NREST-monad. It symbolically executes the abstract program while maintaining refinements for the abstract variables to a concrete representation and generates a concrete program as well as a valid *hnr* predicate. Proof obligations[13] that occur during this process are discharged automatically, guided by user-provided hints where necessary.

The synthesis requires rules for all abstract combinators. For example, `bind` is processed by the following rule:

$$
\begin{array}{ll}
1 & [\![\ \mathit{hnr}\,\Gamma\ m_\dagger\ \Gamma'\ A_x\ m; \\
2 & \quad (\forall x\ x_\dagger.\ \mathit{hnr}\,(A_x\ x_\dagger\ x \star \Gamma')\ (f_\dagger\ x_\dagger)\ (A'_x\ x_\dagger\ x \star \Gamma'')\ A_y\ (f\,x)); \\
3 & \quad \mathit{destructor}\ A'_x\ \mathit{free}\ ]\!] \implies \\
4 & \mathit{hnr}\,\Gamma\ (x_\dagger \leftarrow m_\dagger;\ r_\dagger \leftarrow f_\dagger\ x_\dagger;\ \mathit{free}\ x_\dagger;\ \texttt{return}\ r_\dagger)\ \Gamma''\ A_y\ (x \leftarrow m;\ f\,x)
\end{array}
$$

To refine $x \leftarrow m;\ f\,x$, we first execute m, synthesizing the concrete program $m_\dagger$ (line 1). The state after m is $A_x\ x_\dagger\ x \star \Gamma'$, where x is the result created by m. From this state, we execute $f\,x$ and synthesize $f_\dagger\ x_\dagger$ (line 2). The new state is $A'_x\ x_\dagger\ x \star \Gamma'' \star A_y\ y_\dagger\ y$, where y is the result of $f\,x$. Now, the intermediate variable x goes out of scope and has to be deallocated. The predicate $\mathit{destructor}\ A'_x\ \mathit{free}$ (line 3) states that *free* is a deallocator for data structures implemented by refinement assertion A'_x. Note that *free* can only use time credits that are stored in A'_x. Typically, these are payed for during creation of the data structure (cf. Section 3.6). This way amortization can be used effectively to hide the necessary *free* operation and its costs in the abstract program.

All other combinators ($\texttt{rec}_c$, $\texttt{if}_c$, $\texttt{while}_c$, etc.) have similar rules that are used to decompose an abstract program into parts, synthesize corresponding concrete parts recursively and combine them afterwards with the respective combinators from LLVM. At the leaves of this decomposition, atomic operations need to be provided with suitable synthesis predicates.

An example is a list lookup that is implemented by an array:

$$\mathit{hnr}\,(\mathit{array}_A\ p\ \mathit{xs} \star \mathit{snat}_A\ i_\dagger\ i)$$

[13] E.g. from implementing mathematical integers with fixed-bit machine words.

$$(array\_nth\ p\ i_\dagger)$$
$$(array_A\ p\ xs \star snat_A\ i_\dagger\ i)\ id_A\ (list\_get_{spec}\ xs\ i\ (\lambda\_.\ array\_get_{cost}))$$

Here, the assertions $array_A$, $snat_A$ and id_A relate a list with an array, an unbounded natural number with a bounded signed word and identical elements respectively. With an array at address p holding the list xs and an index $i_\dagger$ that is a bounded signed word representing an unbounded natural number i, *array_nth* leaves the parameters unchanged and extracts the element specified by $list\_get_{spec}$ incurring costs $array\_get_{cost} = \$_{ofs\_ptr} + \$_{load}$.

Ideally, each operation has its own currency (e.g. *list_get*). However, as our definition of *hnr* does not support currency refinement, the basic operations must use the currencies of the LLVM cost model. To still obtain modular *hnr* rules, we encapsulate specifications for data structures with their cost, e.g. by defining $array\_get_{spec} = list\_get_{spec}\ (\lambda\_.\ array\_get_{cost})$. These can easily be introduced in an additional refinement step. Automating this process, and possibly integrating currency refinement into *hnr* is left to future work.

4.3 Notation for Refinement

Synthesis rules typically have the following general form:

$$\begin{aligned} &P\ (x_{1\dagger},\ldots,x_{n\dagger})\ (x_1,\ldots,x_n) \implies \\ &\quad hnr\ (A_1\ x_{1\dagger}\ x_1 \star \ldots \star A_n\ x_{n\dagger}\ x_n) \quad (f_\dagger\ (x_{1\dagger},\ \ldots\ ,x_{n\dagger})) \\ &\qquad (A_1'\ x_{1\dagger}\ x_1 \star \ldots \star A_n'\ x_{n\dagger}\ x_n)\ \ A\ \ (f\ (x_1,\ \ldots\ ,x_n)) \end{aligned}$$

That is, if we have concrete parameters $x_{1\dagger}, \ldots, x_{n\dagger}$ that refine the abstract parameters $x_1, \ldots, x_n$, wrt. refinement assertions $A_1, \ldots, A_n$, and, additionally, the precondition P holds for the parameters, then the result of the concrete function $f_\dagger$ applied to the concrete parameters refines the result of the abstract function applied to the abstract parameters, with assertion A. Moreover, after executing the function, some parameters $x_{i\dagger}$ may still be valid, e.g., if they are only read. In this case, we have $A_i' = A_i$. For parameters that are deleted by the function, or whose ownership is transferred (e.g. into the result), we have $A_i' = del\ A_i$.[14]

We introduce a more succinct notation for synthesis rules of the above form:[15]

$$(f_\dagger, f) \in [P]\ {A_1}^{p_1} \to \ldots \to {A_n}^{p_n} \to A$$

The notation is inspired by relational parametricity rules. The superscripts of the refinement assertions indicate whether the parameter will be kept on the heap ($A_i' = A_i$) or destroyed ($A_i' = del\ A_i$).

Example 4.1. The following expresses the correctness of an implementation $push_\dagger$ of $list\_push_{spec}$:

$$(push_\dagger, list\_push_{spec}\ (t)) \in {L_A}^d \to {E_A}^k \to L_A$$

That is, the first parameter (the list) is refined by the assertion L_A. The $\cdot^d$ annotation expresses that our implementation destructively updates the list, i.e., ownership of the list is transferred into the result. The second parameter (the element) is refined by the assertion E_A. The $\cdot^k$ annotation expresses that our implementation does not change the parameter[16]. Finally, the result list is, again, refined by the assertion L_A.

In Section 5 we will provide such an implementation with dynamic arrays.

[14]Here, $del\ A\ x_\dagger\ x = \uparrow(\exists h.\ A\ x_\dagger\ x\ h)$ just retains the information that the assertion is true for some heap (e.g. the original one). Our framework uses this information to restore the parameter in case the refinement assertion is pure, i.e., does not depend on the heap.

[15]The notation is introduced by Lammich e. g. in [23, §5.1].

[16]Note that this requires the implementation to *copy* the element into the array rather than to just *transfer* its ownership.

4.4 Extracting Hoare Triples

Note that *hnr* predicates cannot always be expressed as Hoare triples, as the running time bound of the abstract program may depend on the result, which we cannot refer to in the precondition of a Hoare triple, where we have to express the allowed running time as time credits.[17] However, if the running time bound does not depend on the result, we can write *hnr* as a Hoare triple:

$$hnr\,\Gamma\ m_\dagger\ \Gamma'\,A\ (\mathsf{spec}\ \Phi\ (\lambda\_.T)) = \{\$T \star \Gamma\}\ m_\dagger\ \{\lambda r.\ \Gamma' \star \exists_A r_a.\ A\ r\ r_a \star \uparrow(\Phi\ r_a)\}$$

While intermediate components might not be of this form, final algorithms typically are. At the end of a development, this rule allows to extract a Hoare triple in the underlying LLVM semantics, cutting out the NREST-monad. For validating the correctness claim of an algorithm, only the final Hoare triple needs to be inspected, which only uses concepts of the underlying semantics.

Note that the above rule is an equivalence. Thus, it can also be used to obtain synthesis rules from Hoare triples provided by the basic VCG infrastructure.

4.5 Attain Supremum

We comment on a problem that arises when composing *hnr* predicates and data refinement in the NREST monad. Consider the following programs and relations:

$m' = \mathtt{res}\ [x \mapsto \$_a,\ y \mapsto \$_b]$ R = {(z, x), (z, y)}
$m = \mathtt{res}\ [z \mapsto \$_a + \$_b]$ A = id_A
$m_\dagger = \mathtt{consume}\ (\$_a + \$_b);\ \mathtt{return}\ z$

Data refinement defines the resource bound for a concrete result (here z) as the supremum over all bounds of related results (here x, y). Thus, we have $m \le \mathtt{res}\ [z \mapsto \$_a + \$_b] = \Downarrow_D R\ m'$. Moreover, we trivially have $hnr\ \square\ m_\dagger\ \square\ A\ m$. Intuitively, we want to compose these two refinements, to obtain $hnr\ \square\ m_\dagger\ \square\ (A \circ R)\ m'$. However, as our definition of *hnr* does not form a supremum, this would require $\$_a + \$_b \le \$_a$ or $\$_a + \$_b \le \$_b$, which obviously does not hold.

We have not yet found a way to define *hnr* or $\Downarrow_D$ in a form that does not exhibit this effect. Instead, we explicitly require that the supremum of the data refinement has a witness. The predicate *attains_sup m m′ R* characterizes that situation: it holds, if for all results r of m the supremum of the set of all abstractions $(r, r') \in R$ applied to m' is in that set. This trivially holds if R is *single-valued*, i.e. any concrete value is related with at most one abstract value, or if m' is *one-time*, i.e. assigns the same resource bound to all its results.

In practice we *do* encounter non-single-valued relations[18], but they only occur as intermediate results where the composition with an *hnr* predicate is not necessary. Also, collapsing synthesis predicates and refinements in the NREST-monad typically is performed for the final algorithm whose running time does not depend on the result, thus is *one-time*, and ultimately *attains_sup*.

5 CASE STUDY: DYNAMIC ARRAYS IN THE ABSTRACT

In this section, we present a case study that shows that amortized data structures can be proven correct on the abstract NREST level. We verify the amortized-constant-time *push* operation of dynamic arrays in the abstract NREST formalism and then synthesize LLVM code from it using the automatic method from the previous section. We focus on the resource consumption and the amortization argument in particular. For presentation purposes we omit functional correctness and

[17] This exact limitation is resolved by the introduction of possibly-negative time credits, described in Guéneau *et al.* [11, 13]. We expect that the *hnr* predicate can be defined with a Hoare triple when using that concept.

[18] The relation *oarr*, described in earlier work [24, 4.2] by one of the authors, is used to model ownership of parts of a list on an abstract level and is an example for a relation that is not single-valued.

some size side conditions that are vital for the implementation in LLVM. We will comment on that towards the end of this section.

5.1 Dynamic Lists

In Example 2.5 we introduced dynamic lists, which model dynamic arrays as a triple of a carrier list, its length and its capacity. We have shown that dl_push_{spec} on dynamic lists refines $list\_push_{spec}$ on lists (Example 2.8). The next step in refining the push operation is to add the abstract algorithmic idea: If we run out of capacity, we double the size of the carrier list and push the element afterwards.

$dl\_push\ (cs,\ l,\ c)\ x =$
$\texttt{if}_c\ l < c\ \texttt{then}$ ($\$_{less}$)
  $dl\_push\_basic_{spec}\ (cs,\ l,\ c)\ x$ ($\$_{dl\_push\_basic}$)
else
  $(cs',\ l',\ c') \leftarrow dl\_double_{spec}\ (cs,\ l,\ c);$ ($\$_{dl\_double_c}$ c)
  $dl\_push\_basic_{spec}\ (cs',\ l',\ c')\ x$ ($\$_{dl\_push\_basic}$)

Here, the program $dl\_push\_basic_{spec}$ pushes an element at the end of the list, assuming that there is enough capacity; and the program dl_double_{spec} doubles the capacity of the dynamic list. The abstract currency *dl_push_basic* represents the costs incurred to push an element and the abstract currency dl_double_c represents the costs to double the dynamic array *per element* in the carrier list.

Let us examine the *raw*, i. e. non-amortized, costs of the operation. If there is capacity left, we have to pay for the `if`-branch and its guard, as well as the basic push operation. This can be sumarized in the constant cost *dl_push* incurrs: $dl\_push\_overhead_{cost} = \$_{less} + \$_{if} + \$_{dl\_push\_basic}$. In the other case, we have to additionally pay for the doubling: $push\_overhead_{cost} + \$_{dl\_double_c}\ c$. Thus, the worst-case cost of the operation is not constant, but rather linear in c because of the *double* operations.

As a next step we will see how we can formalize the potential method on the NREST level and prove that the abstract push operation has amortized constant time.

5.2 Amortized Analysis

The potential method for amortized complexity has the following well-known inequality that relates the raw cost of an operation with its advertised cost and the potential of the data structure before and after an operation.

$$raw\_cost_i \leq (\Phi_i + advertised\_cost_i) - \Phi_{i+1}$$

Before executing an operation we can get the *resource credits* from the potential of the data structure and add it to the cost that is advertised to the caller of the operation. Then, we execute the operation incurring the raw costs, and afterwards we need to give back the *resource credits* for the potential of the resulting data structure. Finally, we can execute several operations on the data structure one after the other and use telescoping to obtain the following inequality

$$\sum_{0 \leq i < n} raw\_cost_i \leq \sum_{0 \leq i < n} advertised\_cost_i$$

Here, we assume that each raw_cost_i and Φ_i is non-negative and the potential Φ_0 is initially zero. The inequality expresses that the real costs are upper bounded by the sum of the advertised costs.

We cannot use `elapse` to model the subtraction in the amortization inequality, as this would require negative costs. Instead, we introduce a new combinator `reclaim` and formulate the amortization inequality in the NREST-monad in the following way:

$$m_{raw}\ ds \leq \texttt{reclaim}\ (\texttt{elapse}\ (m_{adv}\ ds)\ (\Phi\ ds))\ (\lambda ds'.\ \Phi\ ds')$$

Here the raw monadic program m_{raw} executed on some data structure ds has to refine the program that first consumes the potential of the data structure, then executes the monadic program with advertised costs, and in the end reclaims as much costs as the resulting data structure ds' needs for its potential.

The combinator `reclaim` subtracts cost from a monadic program, and fails if it would get negative. Note that this approach only works if the resource type provides a minus operator, as *ecost* does in our case. Here is the formal definition:

$$\texttt{reclaim} :: (\alpha, ecost)\ NREST \to (\alpha \to ecost) \to (\alpha, ecost)\ NREST$$
$$\texttt{reclaim fail}\ T = \texttt{fail}$$
$$\texttt{reclaim (res}\ M)\ T = Sup\ \{\ \texttt{if}\ T\,x \le t'\ \texttt{then res}\ [x \mapsto t' - T\,x]\ \texttt{else fail} \mid t'\,x.\ M\,x = Some\ t'\ \}$$

For each possible result x of M the combinator checks whether the consumed time t' is at least the reclaimed time $T\,x$ for that result. This ensures not falling into the negative when subtracting. If one of the inequalities does not hold, the whole program `reclaim` $m\ t$ fails.

Using `reclaim` we can state the *amortization refinement lemma* for *dl_push*:

$$dl\_push\ (cs, l, c)\ x \le \texttt{reclaim (elapse}\ (dl\_push_{spec}\ push\_adv_{cost}\ (cs, l, c)\ x)\ (\Phi_{dl}\ (cs, l, c)))\ \Phi_{dl}$$

Setting $\Phi_{dl}\ (cs, l, c) = \$_{dl\_double_c}\ (2 * l - c)$ and $push\_adv_{cost} = push\_overhead_{cost} + \$_{dl\_double_c}\ 2$, our VCG can automatically prove this lemma[19].

In particular, we have shown that *dl_push* has amortized constant time, as its advertised cost only consumes the $push\_overhead_{cost}$ and two additional $\$_{dl\_double_c}$ coins for loading the potential. This argument is independent from how exactly *dl_double* is implemented and how the currency $\$_{dl\_double_c}$ is refined later. That way we achieved to separate the amortization argument from the implementation details.

This already concludes the verification on the NREST-level. We have shown that we can use the potential Φ_{dl} to prove *dl_push* having amortized constant time. We can go on proving correct other operations on the data structure with amortization, e. g. lookup, write within bounds, initialization, and destruction. That includes to show that they respect the change of potential. We can also apply telescoping on this level and sequentially compose several `reclaim`–`elapse` pairs on the same data structure following the intuition above.

It is left to show that we can actually implement the operation with a concrete program and obtain the desired synthesis rule mentioned in Example 4.1.

5.3 Moving Potential to Time Credits

Now we have obtained a refinement in the `reclaim`–`elapse` pattern. In order to obtain the desired synthesis rule, we will move the potential from the abstract NREST-program into the pre- and post-heap in the synthesis rule. This will only leave the advertised cost in the abstract program.

On the separation logic level we can augment assertions representing raw data structures with time credits representing their potential. The operator $[\Phi]A\ r\ r_a = \$\Phi\ r_a \star A\ r\ r_a$ adds the potential as time credits depending on the abstract result to an assertion.

Given a synthesis rule that refines a `reclaim`–`elapse` pattern we can move the consumed prepotential into the precondition and the reclaimed postpotential into the assertion of the result.

$$(m_\dagger, \lambda(x, r).\ \texttt{reclaim (consume}\ (m\ x\ r)\ (\Phi\ x))\ \Phi) \in A^d \to {A_R}^k \to A$$
$$\implies\ (m_\dagger, m) \in ([\Phi]A)^d \to {A_R}^k \to [\Phi]A$$

[19]To help us with finding the correct terms for Φ_{dl} and $push\_adv_{cost}$, we can run our VCG with symbolic variables first, and examine the generated proof obligations, which show us the constraints that Φ_{dl} and $push\_adv_{cost}$ must satisfy.

Here, the first parameter (called x in the abstract program) is the amortized data structure that is altered and returned as the result. The second parameter (called r in the abstract program) represents the rest of the parameters. They are not modified in this case and do not contribute with amortized potential. We call this rule an *amortization synthesis rule*. Note that, for simplicity, we have not shown the side conditions that ensure finiteness of the potential and non-failure of the abstract program.

Using that rule the amortization can be moved from the NREST level into the separation logic assertion. The synthesis rule now directly relates the implementation $m_\dagger$ and the monadic program m. In the following we will explain how this is applied to our example.

5.4 Obtaining a Synthesis Rule

In order to obtain a synthesis rule for *list_push*, we first need to provide an implementation and connect it to the program *dl_push*. Observe that *dl_push* lives in the currency system of dynamic lists and not of LLVM currencies. We need to refine it to some abstract program *da_push* that fixes the way we implement the carrier list to arrays and refines all operations to operations we have synthesis rules for. This involves exchanging the currencies from dynamic lists to LLVM currencies via some exchange rate E_{da}. In particular E_{da} has to specify how the coin $\$_{dl\_double_c}$ must be exchanged. Those costs will contain the costs for allocating the new carrier list and copying the elements to the new carrier list. Note that those costs need to be specified *per element* of the original carrier list. For presentation purposes we skip the details of that part and assume we come up with a program *da_push* and a suitable refinement $da\_push\ dl\ x \le \Downarrow_C E_{da}\ (dl\_push\ dl\ x)$.

Furthermore, let da_raw_A be the refinement assertion that relates a concrete representation of a dynamic array with a dynamic list holding natural numbers. While the theory is not dependent on the type of the payload, we choose a fixed one here for presentation purposes. We later want to model strings of characters with the dynamic array. So, the concrete part of the assertion da_raw_A is a triple, consisting of an array of 8 bit integers ($\langle 8\rangle unat_A$) and two 64 bit integers ($\langle 64\rangle snat_A$) for the length and capacity. Further, we assume that we have synthesized an LLVM program $da\_push_\dagger$ that refines *da_push*, with the following synthesis rule:

$$(da\_push_\dagger,\ da\_push) \in (da\_raw_A)^d \to \langle 8\rangle unat_A{}^k \to da\_raw_A$$

Now we can combine the currency refinement rule for *da_push* and the amortization refinement rule for *dl_push* and obtain to the following refinement:

$$da\_push\ dl\ x \le \texttt{reclaim}\ (\texttt{elapse}\ (dl\_push_{spec}\ push\_concrete\_adv_{cost}\ dl\ x)\ (\Phi_{da}\ dl))\ \Phi_{da}$$

Here, the currency refinement was already distributed over `reclaim` and `elapse`. This yields the following two cost functions: $push\_adv'_{cost} = \downarrow_C E_{da}\ push\_adv_{cost}$ and $\Phi_{da}\ dl = \downarrow_C E_{da}\ (\Phi_{dl}\ dl)$. Here, the operation $\downarrow_C E\ t$ applies an exchange rate to a resource function. In particular, as the exchange rate E_{da} is independent of the dynamic list and $push\_adv_{cost}$ is constant, also the advertised cost $push\_adv'_{cost}$ is constant.

We can now combine that refinement rule with the synthesis rule from above. Note that the refinement does not involve data refinement, and thus does not have any *attains_sup* side conditions (cf. Section 4.5). We obtain the following synthesis rule:

$$\begin{aligned}&(\lambda(da,\ x_\dagger).\ da\_push_\dagger\ da\ x_\dagger,\\ &\quad \lambda(dl,\ x).\ \texttt{reclaim}\ (\texttt{elapse}\ (dl\_push_{spec}\ push\_adv'_{cost}\ dl\ x)\ (\Phi_{da}\ dl))\ \Phi_{da})\\ &\in da\_raw_A{}^d \to \langle 8\rangle unat_A{}^k \to da\_raw_A\end{aligned}$$

This form fits the precondition of the amortization synthesis rule, and we can apply it to move the elapsed and reclaimed resources to the pre-heap and the refinement assertion for the result respectively.

$$(\lambda(da, x_\dagger).\ da\_push_\dagger\ da\ x_\dagger,\ \lambda(dl, x)\ dl\_push_{spec}\ push\_adv'_{cost}\ dl\ x)$$
$$\in ([\Phi_{da}]da\_raw_A)^d \to \langle 8\rangle unat_A{}^k \to [\Phi_{da}]da\_raw_A$$

At this point we already have established a refinement between the push operation on dynamic lists dl_push_{spec} and the implementation on dynamic arrays $da\_push_\dagger$. We could extract a Hoare triple from the synthesis rule that shows the correctness of the implementation and the amortized constant running time.

As a last step, we hide the intermediate concept of dynamic lists and obtain a refinement between the list operation and the implementation on dynamic arrays. First, consider the data refinement between dl_push and $list\_push_{spec}$. We repeat it here:

$$(dl\_push_{spec}\ t,\ list\_push_{spec}\ t) \in R^{list}_{dynlist} \to Id \to R^{list}_{dynlist}$$

We can apply this data refinement to the synthesis rule above, and use the fact that $R^{list}_{dynlist}$ is single-valued[20] to solve the sup-attains side condition. Then, we obtain the final synthesis rule:

$$(da\_push_\dagger,\ list\_push_{spec}\ push\_adv'_{cost}) \in da_A{}^d \to \langle 8\rangle unat_A{}^k \to da_A$$

Where da_A relates a list with a dynamic array. This refinement assertion combines the refinement relation $R^{list}_{dynlist}$, the raw refinement assertion da_raw_A and the augmentation with the time credits containing the potential. Formally we define:

$$da_A\ as\ al = \exists_A dl.\ [\Phi_{da}](da\_raw_A)\ dl\ al \star \uparrow((dl, as) \in R^{list}_{dynlist})$$

As mentioned at the beginning of this section, for presentation purposes we have left out size constraints that are necessary to avoid overflows in the LLVM implementation. When doubling the list we have to make sure that the multiplication of the capacity with *2* does not lead to an overflow. We can restrict this by adding a size constraint to the synthesis rule demanding the length of the list may at most be half of *MAX_INT* before pushing an element to it. In a program that uses that operation, one then has to add assertions before those invocations that help the Sepref tool to discharge the respective size constraints. Those size constraints then can be propagated to the precondition of the program. For example, a depth-first search that uses a dynamic array to represent its waiting list might have an additional size constraint restricting the number of edges in the graph to *MAX_INT / 2*.

Once we have the last synthesis rule, we can cut out the whole reasoning with the combinators `reclaim` and `elapse` and inspect the rule on its own. The refinement assertion da_A serves as a black box for the user. For a user of the rule, only the constant advertised cost is visible in $push\_adv'_{cost}$ and the whole amortization is hidden and happens under the hood, such that this amortized data structure behaves like any other data structure.

5.5 Discussion

Previously, we had to prove amortized data structures on the low-level separation logic (e. g. [14, §5.1]), while we can now structure our proofs using the same top-down refinement approach as for non-amortized complexity analysis.

[20] That is, every dynamic list has at most one corresponding abstract list.

While we have demonstrated our method for the quite simple dynamic array data structure, we believe that more involved amortized analyses can also profit from this technique. A next step would be to modularize the verification of Union-Find [6, 28].

Another advantage of performing the analysis on the abstract NREST-level is the independence from the actual back end. E.g., we could[21] use the same abstract proof to verify implementations in LLVM and Imperative HOL.

6 CASE STUDY: INTROSORT

In this section, we apply our framework to the introsort algorithm [30]. We build upon the verification of its functional correctness [24] to verify its running time analysis and synthesize competitive efficient LLVM code for it. Following the "top-down" mantra, we use several intermediate steps to refine a specification down to an implementation.

6.1 Specification of Sorting

We start with the specification of sorting a slice of a list:

$slice\_sort_{spec}\ xs_0\ l\ h\ (t) =$
  `assert` $(l \le h \wedge h \le |xs_0|)$;
  `spec` $(\lambda xs.\ slice\_sort\_aux\ xs_0\ l\ h\ xs)\ (\lambda\_.\ t)$

Where $slice\_sort\_aux\ xs_0\ l\ h\ xs$ states that xs is a permutation of xs_0, xs is sorted between l and h and equal to xs_0 anywhere else.

6.2 Introsort's Idea

The introsort algorithm is based on quicksort. Like quicksort, it finds a pivot element, partitions the list around the pivot, and recursively sorts the two partitions. Unlike quicksort, however, it keeps track of the recursion depth, and if it exceeds a certain value (typically $\lfloor 2 \log n \rfloor$), it falls back to heapsort to sort the current partition. Intuitively, quicksort's worst-case behaviour can only occur when unbalanced partitioning causes a high recursion depth, and the introsort algorithm limits the recursion depth, falling back to the $O(n \log n)$ heapsort algorithm. This combines the good practical performance of quicksort with the good worst-case complexity of heapsort.

Our implementation of introsort follows the implementation of *libstdc++*, which includes a second optimization: a first phase executes quicksort (with fallback to heapsort), but stops the recursion when the partition size falls below a certain threshold τ. Then, a second phase sorts the whole list with one final pass of insertion sort. This exploits the fact that insertion sort is actually faster than quicksort for *almost-sorted* lists, i.e., lists where any element is less than τ positions away from its final position in the sorted list. While the optimal threshold τ needs to be determined empirically, it does not influence the worst-case complexity of the final insertion sort, which is $O(\tau n) = O(n)$ for constant τ. The threshold τ will be an implicit parameter from now on.

While this seems like a quite concrete optimization, the two phases are already visible in the abstract algorithm, which is defined as follows in NREST:

$introsort\ xs\ l\ h =$
  `assert` $(l \le h)$;
  $n \leftarrow$ `return` $h - l$; ($\$_{sub}$)
  `if`$_c$ $n > 1$ `then` ($\$_{lt}$)
    $xs \leftarrow almost\_sort_{spec}\ xs\ l\ h$; ($\$_{almost\_sort}$)

[21] In practice, we have to copy and slightly adjust the proof, as the front-ends for LLVM and Imperative HOL are not yet unified.

```
    xs ← final_sort_spec xs l h                    ($_final_sort)
    return xs
  else return xs
```

Here, $almost\_sort_{spec}\ (t)$ specifies an algorithm that almost-sorts a list, consuming at most t resources and $final\_sort_{spec}\ (t)$ specifies an algorithm that sorts an almost-sorted list, consuming at most t resources.

The program *introsort* leaves trivial lists unchanged and otherwise executes the first and second phase. Its resource usage is bounded by the sum of the first and second phase and some overhead for the subtraction, comparison, and *if-then-else*. Using the verification condition generator we prove that *introsort* is correct, i.e., refines the specification of sorting a slice:

$$introsort\ xs\ l\ h \leq \Downarrow_C E_{is}\ (slice\_sort_{spec}\ xs\ l\ h\ (\$_{sort}))$$

Where $E_{is} = 0(sort := introsort_{cost})$ is the exchange rate used at this step and the total allotted cost for introsort is $introsort_{cost} = \$_{sub} + \$_{if} + \$_{lt} + \$_{almost\_sort} + \$_{final\_sort}$.

6.3 Introsort Scheme

The first phase can be implemented in the following way:

```
1   introsort_aux μ xs l h =
2     d ← depth_spec l h;                          ($_depth)
3     rec_c (λintrosort_rec (xs,l,h,d).
4       assert (l ≤ h);
5       n ← h − l;                                 ($_sub)
6       if_c n > τ then                            ($_lt)
7         if_c d = 0 then                          ($_eq)
8           slice_sort_spec xs l h                 ($_sort_c (μ (h - l)))
9         else
10          (xs, m) ← partition_spec xs l h;       ($_partition_c (h - l))
11          d' ← d − 1;                            ($_sub)
12          xs ← introsort_rec (xs, l, m, d');
13          xs ← introsort_rec (xs, m, h, d');
14          return xs
15      else return xs
16    ) (xs, l, h, d)
```

Where $partition_{spec}$ partitions a slice into two non-empty partitions, returning the start index m of the second partition, and $depth_{spec}$ specifies $\lfloor 2 \log(h - l) \rfloor$.

Let us first analyze the recursive part: if the slice is shorter than the threshold τ, it is simply returned (line 15). Unless the recursion depth limit is reached, the slice is partitioned using $h - l$ $partition_c$ coins, and the procedure is called recursively for both partitions (lines 10-14). Otherwise, the slice is sorted at a price of $\mu\ (h - l)$ $sort_c$ coins (line 8). The function μ here represents the leading term in the asymptotic costs of the used sorting algorithm, and the $sort_c$ coin can be seen as the constant factor. This currency will later be exchanged into the respective currencies that are used by the sorting algorithm. Note that we use currency $sort_c$ to describe costs per comparison of a sorting algorithm, while currency *sort* describes the cost for a whole sorting algorithm.

Showing that the procedure results in an almost-sorted list is straightforward. The running time analysis, however, is a bit more involved. We presume a function μ that maps the length of a slice

to an upper bound on the abstract steps required for sorting the slice. We will later use heapsort with $\mu_{nlogn}\ n = n \log n$.

Consider the recursion tree of a call in *introsort_rec*: We pessimistically assume that for every leaf in the recursion tree we need to call the fallback sorting algorithm. Furthermore, we have to partition at every inner node. This has cost linear in the length of the current slice. For each following inner level the lengths of the slices add up to the current one's, and so do the incurred costs. Finally we have some overhead at every level including the final one. The cost of the recursive part of *introsort_aux* is:

$$\begin{aligned} &introsort\_rec_{cost}\ \mu\ (n, d) = \$_{sort_c}\ (\mu\ n) + \$_{partition_c}\ d * n \\ &\quad + ((d+1)*n)*(\$_{if}\ 2 + \$_{call}\ 2 + \$_{eq} + \$_{lt} + \$_{sub}\ 2) \end{aligned}$$

The correctness of the running time bound is proved by induction over the recursion of *introsort_rec*. If the recursion limit is reached ($d = 0$), the first summand pays for the fallback sorting algorithm. If $d > 0$, part of the second summand pays for the partitioning of the current slice, then the list is split into two and the recursive costs are payed for by parts of all three summands. To bound the costs for the fallback sorting algorithm, μ needs to be *superadditive*: $\mu\ a + \mu\ b \leq \mu\ (a + b)$. In both cases, the third summand pays for the overhead in the current call.

For $d = \lfloor 2 \log n \rfloor$ and an $O(n \log n)$ fallback sorting algorithm ($\mu = \mu_{nlogn}$), $introsort\_rec_{cost}\ \mu_{nlogn}$ is in $O(n \log n)$.[22] In fact, any $d \in O(\log n)$ would do.

Before executing the recursive method, *introsort_aux* calculates the depth limit d. The correctness theorem then reads:

$$introsort\_aux\ \mu_{nlogn}\ xs\ l\ h \leq \Downarrow_C(E_{isa}(h - l))(almost\_sort_{spec}\ xs\ l\ h\ \$_{almost\_sort})$$

Where $E_{isa}\ n = 0(almost\_sort := \$_{depth} + introsort\_rec_{cost}\ \mu_{nlogn}\ (n, \lfloor 2 \log n \rfloor))$.

Note that specifications typically use a single coin of a specific currency for their abstract operation, which is then exchanged for the actual costs, usually depending on the parameters.

This concludes the interesting part of the running time analysis of the first phase. It is now left to plug in an $O(n \log n)$ fallback sorting algorithm, and a linear partitioning algorithm.

Heapsort. Independently of introsort, we have proved correctness and worst-case complexity of heapsort, yielding the following refinement lemma:

$$heapsort\ xs\ l\ h \leq \Downarrow_C(E_{hs}\ (h - l))\ (slice\_sort_{spec}\ xs\ l\ h\ (\$_{sort}))$$

Where $E_{hs}\ n = 0(sort := c_1 + log\ n * c_2 + n * c_3 + (n * log\ n) * c_4)$ for some constants $c_i :: ecost$.

Assuming that $n \geq 2$,[23] we can estimate $E_{hs}\ n\ sort \leq \mu_{nlogn}\ n * c$, for $c = c_1 + c_2 + c_3 + c_4$, and thus get, for $E_{hs'} = 0(sort_c := c)$:

$$\begin{aligned} &\Downarrow_C(E_{hs}\ (h - l))\ (slice\_sort_{spec}\ xs\ l\ h\ (\$_{sort})) \\ &\leq \Downarrow_C E_{hs'}\ (slice\_sort_{spec}\ xs\ l\ h\ (\$_{sort_c}\ (\mu_{nlogn}\ (h - l)))) \end{aligned}$$

and, by, transitivity

$$heapsort\ xs\ l\ h \leq \Downarrow_C E_{hs'}\ (slice\_sort_{spec}\ xs\ l\ h\ (\$_{sort_c}\ (\mu_{nlogn}\ (h - l))))$$

Note that our framework allowed us to easily convert the abstract currency from a single operation-specific *sort* coin to a $sort_c$ coin for each comparison operation.

[22] More precisely, the sum over all (finitely many) currencies is in $O(n \log n)$.

[23] Note that this is a valid assumption, as heapsort will never be called for trivial slices.

Partition and Depth Computation. We implement partitioning with the Hoare partitioning scheme using the median-of-3 as the pivot element. Moreover, we implement the computation of the depth limit ($2\lfloor\log(h-l)\rfloor$) by a loop that counts how often we can divide by two until zero is reached. This yields the following refinement lemmas:

$$pivot\_partition\ xs\ l\ h \le \Downarrow_C E_{pp}\ (partition_{spec}\ xs\ l\ h\ (\$_{partition_c}\ (h-l)))$$
$$calc\_depth\ l\ h \le \Downarrow_C (E_{cd}\ (h-l))\ (depth_{spec}\ l\ h\ (\$_{depth}))$$

Combining the Refinements. We replace $slice\_sort_{spec}$, $partition_{spec}$ and $depth_{spec}$ by their implementations *heapsort*, *pivot_partition* and *calc_depth*. Finally, we call the resulting implementation $introsort\_aux_2$, and prove

$$introsort\_aux_2\ xs\ l\ h \le \Downarrow_C (E_{aux}\ (h-l))\ (introsort\_aux\ \mu_{nlogn}\ xs\ l\ h)$$

Where the exchange rate E_{aux} combines the exchange rates $E_{hs'}$, E_{pp} and E_{cd} for the component refinements.

Transitive combination with the correctness lemma for *introsort_aux* then yields the correctness lemma for $introsort\_aux_2$:

$$introsort\_aux_2\ xs\ l\ h \le \Downarrow_C (E_{isa2}\ (h-l))\ (almost\_sort_{spec}\ xs\ l\ h\ (\$_{almost\_sort}))$$

Where $E_{isa2}\ n = 0 (almost\_sort := \downarrow_C (E_{aux}\ n)\ (introsort\_aux_{cost}\ n))$ and the operation $\downarrow_C E\ t$ applies an exchange rate to a resource function.

Refining Resources. The stepwise refinement approach allows to structure an algorithm verification in a way that correctness arguments can be conducted on a high level and implementation details can be added later. Resource currencies permit the same for the resource analysis of algorithms: they summarize compound costs, allow reasoning on a higher level of abstraction and can later be refined into fine-grained costs. For example, in the resource analysis of *introsort_aux* the currencies $sort_c$ and $partition_c$ abstract the cost of the respective subroutines. The abstract resource argument is independent from their implementation details, which are only added in a subsequent refinement step, via the exchange rate E_{aux}.

6.4 Final Insertion Sort

The second phase is implemented by insertion sort, repeatedly calling the subroutine *insert*. The specification of *insert* for an index i captures the intuition that it goes from a slice that is sorted up to index $i-1$ to one that is sorted up to index i. Insertion is implemented by moving the last element to the left, as long as the element left of it is greater (or the start of the list has been reached). Moving an element to its correct position takes at most τ steps, as after the first phase the list is almost-sorted, i.e., any element is less than τ positions away from its final position in the sorted list. Moreover, elements originally at positions greater τ will never reach the beginning of the list, which allows for the *unguarded* optimization. It omits the bounds check for those elements, saving one index comparison in the innermost loop. Formalizing these arguments yields the implementation *final_insertion_sort* that satisfies

$$final\_insertion\_sort\ xs\ l\ h \le \Downarrow_C (E_{fis}(h-l))\ (final\_sort_{spec}\ xs\ l\ h\ (\$_{final\_sort}))$$

Where $E_{fis}\ n = 0 (final\_sort := final\_insertion_{cost}\ n)$, and $final\_insertion_{cost}\ n$ is linear in n.

Note that *final_insertion_sort* and $introsort\_aux_2$ use the same currency system. Plugging both refinements into *introsort* yields $introsort_2$ and the lemma

$$introsort_2\ xs\ l\ h \le \Downarrow_C (E_{is2}(h-l))\ (introsort\ xs\ l\ h)$$

Where the exchange rate E_{is2} combines the rates E_{isa2} and E_{fis}.

6.5 Separating Correctness and Complexity Proofs

A crucial function in heapsort is *sift_down*, which restores the heap property by moving the top element down in the heap. To implement this function, we first prove correct a version $sift\_down_1$, which uses swap operations to move the element. In a next step, we refine this to $sift\_down_2$, which saves the top element, then executes upward moves instead of swaps, and, after the last step, moves the saved top element to its final position. This optimization spares half of the memory accesses, exploiting the fact that the next swap operation will overwrite an element just written by the previous swap operation.

However, this refinement is not structural: it replaces swap operations by move operations, and adds an additional move operation at the end. At this point, we chose to separate the functional correctness and resource aspect, to avoid the complexity of a combined non-structural functional and currency refinement. It turns out that proving the complexity of the optimized version $sift\_down_2$ directly is straightforward. Thus, as sketched in Section 2.5, we first prove[24] $sift\_down_2 \le sift\_down_1 \le sift\_down_{spec}\ (\infty)$, ignoring the resource aspect. Separately, we prove $sift\_down_2 \le_n \mathsf{spec}\ (\lambda\_.\ True)\ sift\_down_{cost}$, and combine the two statements to get the final refinement lemma:

$$sift\_down_2 \le sift\_down_{spec}\ sift\_down_{cost}$$

6.6 Refining to LLVM

To obtain an LLVM implementation of our sorting algorithm, we have to specify an implementation for the data structure that holds the elements, and for the comparison operator on elements. We use arrays for the data structure, and parameterize over the comparison function (see Section 6.7). Let E_3 be the corresponding exchange rate from abstract data structure access and comparison to actual LLVM operations. We obtain $introsort_3\ xs\ l\ h \le \Downarrow_C E_3\ (introsort_2\ xs\ l\ h)$, and can automatically synthesize an LLVM program $introsort_\dagger$ that refines $introsort_3$, i.e., satisfies the theorem:

$$(introsort_\dagger,\ introsort_3) \in {array_A}^d \to {snat_A}^k \to {snat_A}^k \to array_A$$

Combination with the refinement lemmas for $introsort_3$, $introsort_2$, and *introsort*, followed by conversion to a Hoare triple, yields our final correctness statement:

$$\begin{array}{l}
[\![\ l \le h;\ h < |xs_0|\ ]\!] \implies \\
\{\$(introsort_{\dagger cost}\ (h-l)) \star array_A\ p\ xs_0 \star snat_A\ l_\dagger\ l \star snat_A\ h_\dagger\ h\} \\
\quad introsort_\dagger\ p\ l_\dagger\ h_\dagger \\
\{\lambda r.\ \exists_A xs.\ array_A\ r\ xs \star \uparrow(slice_sort_aux\ xs_0\ l\ h\ xs) \star snat_A\ l_\dagger\ l \star snat_A\ h_\dagger\ h\}
\end{array}$$

Where $introsort_{\dagger cost} :: nat \to ecost$ is the cost bound obtained from applying the exchange rates E_{is}, then E_{is2}, and finally E_3 to $\$_{sort}$.

Note that this statement is independent of the Refinement Framework. Thus, to believe in its meaningfulness, one has to only check the formalization of Hoare triples, separation logic, and the LLVM semantics.

To formally prove the statement "$introsort_\dagger$ *has complexity* $O(n \log n)$", we first observe that $introsort_{\dagger cost}$ uses only finitely many currencies, and only finitely many coins of each currency. Then, we define the overall number of coins as

$$introsort_{\dagger allcost}\ n = \Sigma c.\ introsort_{\dagger cost}\ n\ c$$

which expands to

[24] Note that we have omitted the function parameters for better readability.

$$introsort_{\dagger\, allcost}\ n = 4693 + 5 * \ log\ n + 231 * n + 455 * (n * log\ n)$$

which, in turn, is routinely proved to be in $O(n \log n)$.

Finally, instantiating the element type and comparison operation yields a complete LLVM program, that our code generator can translate to actual LLVM text and a corresponding header file for interfacing our sorting algorithm from C or C++. For example, with LLVM's *i64* type and the unsigned compare operation *ll_icmp_ult*, we get a program that sorts unsigned 64 bit integers in ascending order.

As LLVM does not support generics, we cannot implement a replacement for C++'s generic *std::sort*. However, by repeating the instantiation for different types and compare operators, we can implement a replacement for any fixed element type.

6.7 Sorting Strings

We now elaborate on the parameterization over element types that we described in the last section, and also show how to sort elements with non-constant-time compare operations, such as strings.

To parameterize over the element type, we define the $introsort_3$ and $introsort_\dagger$ functions inside a *locale* that fixes the relevant parameters:

locale *sort_impl_context* = . . .
  fixes $(<_\dagger) :: \alpha_\dagger \to \alpha_\dagger \to 1\ word\ M$
    and $c :: ecost$
    and $A :: \alpha \to \alpha_\dagger \to assn$
  assumes $((<_\dagger), \texttt{consume}\ c\ (\texttt{return}\ oo\ (<))) \in A^k \to A^k \to bool1_A$
    and *finite_cost c*
. . .

Here, α is the abstract element type, $\alpha_\dagger$ is the concrete element type, $<_\dagger$ is the implementation of the compare function that requires cost c, and A is the refinement relation for elements. The assumptions state that $<_\dagger$ actually implements the comparison, and that the required costs are finite.

This locale can now be instantiated for different element types. For example, the instantiation to *uint64*—as described in the previous section—is done as follows:

global_interpretation *sort_impl_context* . . . *ll_icmp_ult* $\$_{icmp\_ult}\ \langle 64\rangle unat_A$

A more complex element datatype is string. It can be implemented by dynamic arrays[25] (cf. Section 5). In the original formalization without costs, it is straightforward to implement a lexicographic compare operator on dynamic arrays ($strcmp_\dagger$), show that it refines the lexicographic ordering on lists, and instantiate the parameterized sorting algorithm.

However, when adding costs, the costs of comparing two strings depend on the lengths of the strings. In our implementation, comparison is linear in the length of the shorter string. This dependency on the input parameters poses a challenge to the analysis of the algorithm. In our formalization, we simply over-estimate the cost for a comparison by the longest string in the array to be sorted. While more precise analyses might be possible, this approach integrates nicely into our existing formalization infrastructure, and still yields usable upper bounds for not too extreme length distributions.

[25]In C++, the string datatype is typically implemented by a dynamic array, too, however, with some optimizations for short strings, which we omit here.

To integrate our over-estimation into the existing formalization, we define an element assertion that contains a maximum length parameter N, constraining the length of the strings in the array to at most N:

$$bstring_A\ N = bound_A\ (da_A)\ (\lambda xs.\ |xs| < N)$$

Here, the assertion $bound_A\ A\ P\ c\ a = A\ c\ a \star \uparrow(P\ a)$ restricts an assertion A by a predicate P on the abstract values.

Using this assertion, we can estimate the cost of a string comparison ($strcmp_c\ N$) to only depend on N, and instantiate the algorithm as follows:

global_interpretation *sort_impl_context* ... $strcmp_\dagger\ (strcmp_c\ N)\ (bstring_A\ N)$ **for** N

While this instantiation is still parametric in N, the parameter N does not occur in the implementation, such that we get a fully instantiated implementation which we can export to actual LLVM text. In the final correctness statement, the costs are parameterized over N, and we get the estimation:

$$introsort_{\dagger allcost}\ N\ n \in O(N * n * \log n)$$

Discussion. Thanks to Isabelle's locale mechanism, instantiation of our algorithm to an element relation that depends on an extra parameter is pretty straightforward, thus allowing us to also estimate running times for element types with more complex comparison functions, like strings.

Instead of refining the abstract currency for comparing elements to a parametric currency, and then further instantiating the parameters with a concrete implementation, we could also have done the instantiation to element types on the abstract level, and then refined the algorithm to LLVM for each element type. However, our parametric approach saves the overhead of duplicating these refinement steps for each element type.

6.8 Benchmarks

In this section we present benchmarks comparing the code extracted from our formalization with the real world implementation of introsort from the GNU C++ Library (*libstdc++*). Also, as a regression test, we compare with the code extracted from an earlier formalization of introsort [24] that did not verify the running time complexity and used an earlier iteration of the Sepref framework and LLVM semantics without time.

Ideally, the same algorithm should take exactly the same time when repeatedly run on the same data and machine. However, in practice, we encountered some noise up to 17%. Thus, we have repeated each experiment at least ten times, and more often to confirm outliers where the verified and unverified algorithms' run times differ significantly. Assuming that the noise only slows down an algorithm, we take the fastest time measured over all repetitions. The results are shown in Figure 1. As expected, all three implementations have similar running times. We conclude that adding the complexity proof to our introsort formalization, and the time aspect to our refinement process has not introduced any timing regressions in the generated code. Note, however, that the code generated by our current formalization is not identical to what the original formalization generated. This is mainly due to small changes in the formalization introduced when adding the timing aspect.

7 CONCLUSIONS

We have presented a refinement framework for the simultaneous verification of functional correctness and complexity of algorithm implementations with competitive practical performance.

We use stepwise refinement to separate high-level algorithmic ideas from low-level optimizations, enabling convenient verification of highly optimized algorithms. The novel concept of resource

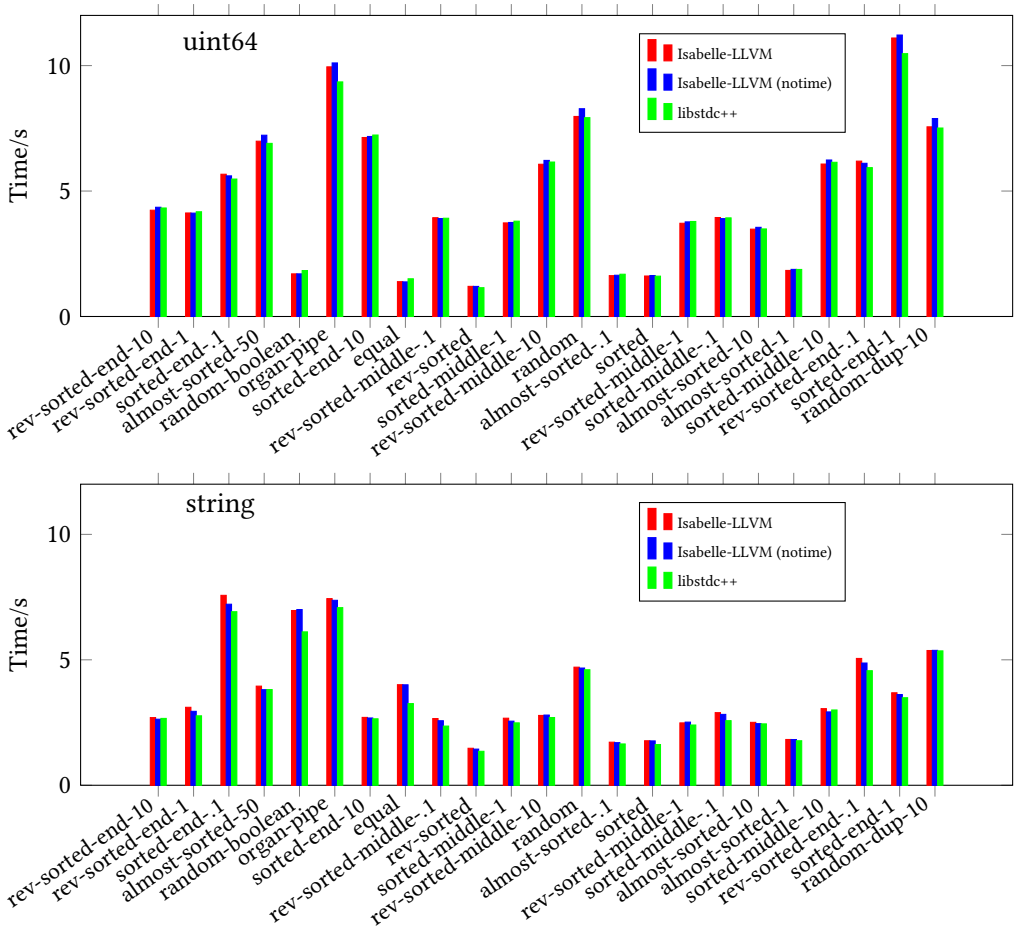

Fig. 1. Comparison of the running time measured for the code generated by the formalization described in this paper (Isabelle-LLVM), the original formalization from [24] (notime), and the *libstdc++* implementation. Arrays with 10^8 *uint64*s and 10^7 *string*s with various distributions were sorted, and we display the smallest time of 10 runs. The programs were compiled with *clang-10 -O3*, and run on an Intel XEON E5-2699 with 128GiB RAM and 256K/55M L2/L3 cache.

currencies allows structuring of the complexity proofs along the refinement chain. Refinement also works seamlessly for amortized data structures. Our framework refines down to the LLVM intermediate representation, such that we can use a state-of-the-art compiler to generate performant programs.

As a case study, we have proved the functional correctness and complexity of the introsort sorting algorithm. Our design supports arbitrary element types, even those with non-constant-time compare operations, like strings. Our verified implementation performs on par with the (unverified) state-of-the-art implementation from the GNU C++ Library. It also provably meets the C++11 standard library [8] specification for *std::sort*, which in particular requires a worst-case time complexity of $O(n \log n)$. We are not aware of any other verified implementations of real-world sorting algorithms that come with a complexity analysis.

Our work is a combination and substantial extension of an earlier refinement framework for functional correctness [22] which also comes with a verification of introsort [24], and a refinement framework for a single *enat*-valued currency [14]. In particular, we have generalized the refinement framework to arbitrary resources, applied it to amortized analysis, introduced currencies that help organizing refinement proofs, extended the LLVM semantics and reasoning infrastructure with a cost model, connected it to the refinement framework via a new version of the Sepref tool, and, finally, added the complexity analysis for introsort.

7.1 Related Work

Nipkow *et al.* [31, §4.1] collect verification efforts concerning sorting algorithms. We add a few instances verifying running time: Wang *et al.* use TiML [36] to verify correctness and asymptotic time complexity of mergesort automatically. Zhan and Haslbeck [37] verify functional correctness and asymptotic running time analysis of imperative versions of insertion sort and mergesort. We build on earlier work by Lammich [24] and provide the first verification of functional correctness and asymptotic running time analysis of heapsort and introsort.

The idea to generalize the nres monad [26] to resource types originates from Carbonneaux *et al.* [4]. They use potential functions ($state \to enat$) instead of predicates ($state \to bool$), present a quantitative Hoare logic, and extend the CompCert compiler to preserve properties of stack-usage from programs in Clight to compiled programs. Observe, that the step from qualitative [9] to quantitative weakest preconditions (cf. Section 2.5) is similar to the weakest preexpectation transformer by Kozen [18], and the expected running time transformer *ert* by Kaminski *et al.* [17].

Rajani *et al.* [33] present a unifying type-theory λ^{amor} for higher-order amortized cost analysis, which involves a cost monad similar to NREST without nondeterminism. The introduction of the `elapse` combinator is straightforward, but the `reclaim` operator in NREST seems to be related to their type constructor $[p]\tau$. That constructor is central to their paper. Rajani [32] applies type-theoretic approach to Information Flow Control and generalizes the theory to allow any commutative monoid in the cost monad. It would be interesting to see whether their cost monad can be extended to nondeterminism.

We see our paper in the line of research concerning simultaneously verifying functional correctness and worst-case time complexity of algorithms. Atkey [1] pioneered resource analysis with separation logic. Charguéraud and Pottier [6, 7] present a framework that uses time credits in Coq and apply it to the Union-Find data structure. Guéneau *et al.* extend that framework with big-O style specifications [12] and possibly negative time credits, and apply it to involved algorithms and data structures [13]. We further develop their work in three ways: First, while time credits usually are natural numbers [1, 7, 12, 29, 37] or integers [13], we generalize to an abstract resource type and specifically use resource currencies for a fine-grained analysis. Second, we use stepwise refinement to structure the verification and make the resource analysis of larger use-cases manageable. Third, we provide facilities to automatically extract efficient competitive code from the verification.

The following are the most complex algorithms and data structures with verified running time analysis using time credits and separation logic we are aware of: a linear time selection algorithm [37], an incremental cycle detection algorithm [13], Union-Find [7], Edmonds-Karp and Kruskal's algorithm [14].

7.2 Future Work

A verified compiler down to machine code would further reduce the trusted code base of our approach. While that is not expected to be available soon for LLVM in Isabelle, the NREST-monad and the Sepref tool are general enough to connect to a different back end. Formalizing one of the

CompCert C semantics [2] in Isabelle, connecting it to the NREST-monad and then processing synthesized C code with CompCert's verified compiler would be a way to go.

In this paper we apply our framework to verify an involved algorithm that only uses basic data structures, i.e. arrays. A next step is to verify more involved data structures, e.g. by porting existing verifications of the Imperative Collections Framework [23] to LLVM. We do not yet see how to reason about the running time of data structures like hash maps, where worst-case analysis would be possible but not useful. In general, extending the framework to average-case analysis and probabilistic programs are exciting roads to take.

We plan to implement more automation, saving the user from writing boilerplate code when handling resource currencies and exchange rates.

Neither the LLVM nor the NREST level of our framework is tied to running time. Applying it to other resources like maximum heap space consumption might be a next step.

ACKNOWLEDGMENTS

We thank Armaël Guéneau, Arthur Charguéraud, François Pottier, and the anonymous referees of ESOP2021 who provided valuable feedback on an earlier version of this paper. This work was supported by the DFG Koselleck grant NI 491/16-1 "Verifizierte Algorithmenanalyse" and the DFG grant LA 3292/1 "Verifizierte Model Checker".

REFERENCES

[1] Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *European Symposium on Programming, ESOP 2010 (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 85–103. https://doi.org/10.1007/978-3-642-11957-6_6

[2] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reason.* 43, 3 (2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[3] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Symposium on Logic in Computer Science (LICS 2007)*. IEEE Computer Society, 366–378. https://doi.org/10.1109/LICS.2007.30

[4] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 270–281. https://doi.org/10.1145/2594291.2594301

[5] Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 116:1–116:34. https://doi.org/10.1145/3408998

[6] Arthur Charguéraud and François Pottier. 2015. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference, ITP 2015 (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 137–153. https://doi.org/10.1007/978-3-319-22102-1_9

[7] Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

[8] cppreference. [n.d.]. C++ standard library specification of sort. https://en.cppreference.com/w/cpp/algorithm/sort. Accessed: 2020-10-12.

[9] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. https://www.worldcat.org/oclc/01958445

[10] GNU C++ Library [n.d.]. The GNU C++ Library. https://gcc.gnu.org/onlinedocs/libstdc++/ Version 7.4.0.

[11] Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes).* Ph.D. Dissertation. Inria, Paris, France. https://tel.archives-ouvertes.fr/tel-02437532

[12] Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018 (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

[13] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *10th International Conference on Interactive Theorem Proving, ITP 2019 (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:20. https://doi.org/10.4230/LIPIcs.ITP.2019.18

[14] Maximilian P. L. Haslbeck and Peter Lammich. 2019. Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving, ITP 2019 (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:18. https://doi.org/10.4230/LIPIcs.ITP.2019.20

[15] Maximilian P. L. Haslbeck and Peter Lammich. 2021. For a Few Dollars More - Verified Fine-Grained Algorithm Analysis Down to LLVM. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 292–319. https://doi.org/10.1007/978-3-030-72019-3_11

[16] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321–. https://doi.org/10.1145/366622.366644

[17] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. 2016. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming Languages and Systems*. Springer, 364–389.

[18] Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. Syst. Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

[19] Alexander Krauss. 2010. Recursive Definitions of Monadic Functions. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010 (EPTCS, Vol. 43)*, Ana Bove, Ekaterina Komendantskaya, and Milad Niqui (Eds.). 1–13. https://doi.org/10.4204/EPTCS.43.1

[20] Peter Lammich. 2015. Refinement to Imperative/HOL. In *Interactive Theorem Proving - 6th International Conference, ITP 2015 (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 253–269. https://doi.org/10.1007/978-3-319-22102-1_17

[21] Peter Lammich. 2016. Refinement based verification of imperative data structures. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 27–36. https://doi.org/10.1145/2854065.2854067

[22] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving, ITP 2019 (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

[23] Peter Lammich. 2019. Refinement to Imperative HOL. *J. Autom. Reason.* 62, 4 (2019), 481–503. https://doi.org/10.1007/s10817-017-9437-1

[24] Peter Lammich. 2020. Efficient Verified Implementation of Introsort and Pdqsort. In *IJCAR 2020 (Lecture Notes in Computer Science, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 307–323. https://doi.org/10.1007/978-3-030-51054-1_18

[25] Peter Lammich and Rene Meis. 2012. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs* (Nov. 2012). http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

[26] Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *Interactive Theorem Proving - Third International Conference, ITP 2012 (Lecture Notes in Computer Science, Vol. 7406)*, Lennart Beringer and Amy P. Felty (Eds.). Springer, 166–182. https://doi.org/10.1007/978-3-642-32347-8_12

[27] libc++ [n.d.]. "libc++" C++ Standard Library. https://libcxx.llvm.org/

[28] Adrián Löwenberg Casas. 2019. *Proof af the Amortized time complexity of an efficient Union-Find data structure in Isabelle/HOL.* BS Thesis. Technical University of Munich.

[29] Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019 (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

[30] David R. Musser. 1997. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* 27, 8 (1997), 983–993.

[31] Tobias Nipkow, Manuel Eberl, and Maximilian P. L. Haslbeck. 2020. Verified Textbook Algorithms - A Biased Survey. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020 (Lecture Notes in Computer Science, Vol. 12302)*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer, 25–53. https://doi.org/10.1007/978-3-030-59152-6_2

[32] Vineet Rajani. 2020. *A type-theory for higher-order amortized analysis.* Ph.D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29104

[33] Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434308

[34] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

[35] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 61âĂŞ78. https://doi.org/10.1145/91556.91592

[36] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 79:1–79:26. https://doi.org/10.1145/3133903

[37] Bohua Zhan and Maximilian P. L. Haslbeck. 2018. Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018 (Lecture Notes in Computer Science, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 532–548. https://doi.org/10.1007/978-3-319-94205-6_35