Verified Memoization and Dynamic Programming

Simon Wimmer, Shuwei Hu, and Tobias Nipkow

Fakultät für Informatik, Technische Universität München

Abstract. We present a lightweight framework in Isabelle/HOL for the automatic verified (functional or imperative) memoization of recursive functions. Our tool constructs a memoized version of the recursive function and proves a correspondence theorem between the two functions. A number of simple techniques allow us to achieve bottom-up computation and space-efficient memoization. The framework's utility is demonstrated on a number of representative dynamic programming problems.

1 Introduction

Verification of functional properties of programs is most easily performed on functional programs. Performance, however, is more easily achieved with imperative programs. One method of improving performance of functional algorithms automatically is memoization. In particular dynamic programming is based on memoization. This paper presents a framework and a tool [24] (for Isabelle/HOL [16, 17]) that memoizes pure functions automatically and proves that the memoized function is correct w.r.t. the original function. Memoization is parameterized by the underlying memory implementation which can be purely functional or imperative. We verify a collection of representative dynamic programming algorithms at the functional level and derive efficient implementations with the help of our tool. This appears to be the first tool that can memoize recursive functions (including dynamic programming algorithms) and prove a correctness theorem for the memoized version.

1.1 Related Work

Manual memoization has been used in specific projects before, e.g. [21, 3], but this did not result in an automatic tool. One of the few examples of dynamic programming in the theorem proving literature is a formalization of the CYK algorithm where the memoizing version (using HOL functions as tables) is defined and verified by hand [2]. In total it requires 1000 lines of Isabelle text. Our version in Section 3.4 is a mere 120 lines and yields efficient imperative code.

Superficially very similar is the work by Itzhaky *et al.* [9] who present a system for developing optimized DP algorithms by interactive, stepwise refinement focusing on optimizations like parallelization. Their system contains an ad-hoc logical infrastructure powered by an SMT solver that checks the applicability

conditions of each refinement step. However, no overall correctness theorem is generated and the equivalent of our memoization step is performed by their backend, a compiler to C++ which is part of the trusted core.

We build on existing infrastructure in Isabelle/HOL for generating executable code in functional and imperative languages automatically [1, 7, 6].

2 Memoization

2.1 Overview

The workhorse of our framework is a tool that automatically memoizes [15] recursive functions defined with Isabelle's function definition command [11]. More precisely, to memoize a function f, the idea is to pass on a memory between invocations of f and to check whether the value of f for x can already be found in the memory whenever $f\,x$ is to be computed. If $f\,x$ is not already present in the memory, we compute $f\,x$ using f's recursive definition and store the resulting value in the memory. The memory is threaded through with the help of a *state monad*. Starting from the defining equations of f, our algorithm produces a version f'_{m} that is defined in the state monad. The only place where the program actually interacts with the state is on recursive invocations of f'_{m}. Each defining equation of f of the form

$$f\,x = t$$

is re-written to

$$f'_{\mathsf{m}}\,x =_{\mathsf{m}} t_{\mathsf{m}}$$

where t_{m} is a version of t defined in the state monad. The operator $=_{\mathsf{m}}$ encapsulates the interaction with the state monad. Given x, it checks whether the state already contains a memoized value for $f\,x$ and returns that or runs the computation t_{m} and adds the computed value to the state. Termination proofs for f'_{m} are replayed from the termination proofs of f. To prove that f still describes the same function as f'_{m}, we use relational parametricity combined with induction. The following subsections will explain in further detail each of the steps involved in this process: *monadification* (i.e. defining f'_{m} in the state monad), replaying the termination proof, proving the correspondence of f and f'_{m} via relational parametricity, and implementing the memory. Moreover, we will demonstrate how this method can be adopted to also obtain a version f_{h} that is defined in the *heap monad* of Imperative HOL [4] and allows one to use imperative implementations of the memory.

2.2 Monadification

We define the state monad with memory of type $'m$ and pure (result) type $'a$:

$$\textbf{datatype}\ ('m,\ 'a)\ \mathit{state} = \mathit{State}\ (\mathit{run\_state} :\ 'm \to 'a \times 'm)\ .$$

That is, given an initial state of type $'m$, a computation in the state monad produces a pair of a computation result of type $'a$ and a result state (of type

$'m$). To make type $('m,\ 'a)\ state$ a monad, we need to define the operators *return* ($\langle$-$\rangle$) and *bind* ($\gg\!=$):

$$\begin{aligned}&return :: 'a \to ('m,\ 'a)\ state\\ &bind :: ('m,\ 'a)\ state \to ('a \to ('m,\ 'b)\ state) \to ('m,\ 'b)\ state\end{aligned}$$

$$\begin{aligned}\langle a\rangle &= State\ (\lambda M.\ (a, M))\\ s \gg\!= f &= State\ (\lambda M.\ \mathsf{case}\ run\_state\ s\ M\ \mathsf{of}\ (a, M') \Rightarrow run\_state\ (f\ a)\ M')\end{aligned}$$

The definition of $\gg\!=$ describes how states are threaded through the program.

There are a number of different styles of turning a purely functional program into a corresponding *monadified* version (see e.g. [5]). We opt for the call-by-value monadification style from [8]. This style of monadification has two distinct features: it fixes a strict call-by-value monadification order and generalizes suitably to higher-order functions. The type $M(\tau)$ of the monadified version of a computation of type τ can be described recursively as follows:

$$\begin{aligned}M(\tau) &= ('m, M'(\tau))\ state\\ M'(\tau_1 \to \tau_2) &= M'(\tau_1) \to M(\tau_2)\\ M'(\tau_1 \oplus \tau_2) &= M'(\tau_1) \oplus M'(\tau_2) && \text{where } \oplus \in \{+, \times\}\\ M'(\tau) &= \tau && \text{otherwise}\end{aligned}$$

As a running example, consider the *map* function on lists. Its type is

$$('a \to 'b) \to ('a\ list \to 'b\ list)$$

and its monadified version map_{m} has type

$$('m,\ ('a \to ('m, 'b)\ state) \to ('m, 'a\ list \to ('m, 'b\ list)\ state)\ state)\ state\ .$$

The definitions of map_{m} and map'_{m} are

$$\begin{aligned}map_{\mathsf{m}} &= \langle \lambda f'_{\mathsf{m}}.\ \langle \lambda xs.\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ xs\rangle\rangle\\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ [\,] &= \langle [\,]\rangle\\ map'_{\mathsf{m}}\ f'_{\mathsf{m}}\ (Cons\ x\ xs) &= Cons_{\mathsf{m}} \bullet (\langle f'_{\mathsf{m}}\rangle \bullet \langle x\rangle) \bullet (map'_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}}\rangle \bullet \langle xs\rangle)\\ Cons_{\mathsf{m}} &= \langle \lambda\ x.\ \langle \lambda\ xs.\ \langle Cons\ x\ xs\rangle\rangle\rangle\end{aligned}$$

compared to *map*:

$$\begin{aligned}map\ f\ [\,] &= [\,]\\ map\ f\ (Cons\ x\ xs) &= Cons\ (f\ x)\ (map\ f\ xs)\ .\end{aligned}$$

As can be seen in this definition, the idea of the translation is to wrap up all bound variables in a *return*, to replace function application with the "$\bullet$"-operator (see below), and to replace constants by their corresponding monadified versions. The combinator map'_{m} follows the recursive structure of *map* and combinator map_{m} lifts map'_{m} from type $M'(\tau)$ to $M(\tau)$ where τ is the type of *map*.

The lifted function application operator "$\bullet$" simply uses $\gg\!=$ to pass along the state:

$$f_{\mathsf{m}} \bullet x_{\mathsf{m}} = f_{\mathsf{m}} \gg\!= (\lambda f.\ x_{\mathsf{m}} \gg\!= f)\ .$$

Our monadification algorithm can be seen as a set of rewrite rules, which are applied in a bottom-up manner. The algorithm will maintain a mapping Γ from terms to their corresponding monadified versions. If we monadify an equation of the form $f\,x = t$ into $f'_{\mathsf{m}}\,x'_{\mathsf{m}} = t_{\mathsf{m}}$ [1], then the initial mapping Γ_0 includes the bindings $f \mapsto \langle f'_{\mathsf{m}}\rangle$ and $x \mapsto \langle x'_{\mathsf{m}}\rangle$. Let $\Gamma \vdash t \rightsquigarrow t_{\mathsf{m}}$ denote that t is monadified to t_{m} given Γ. Moreover, we say that a term is *Γ-pure* if none of its subterms are in the domain of Γ. Our monadification rules are the following:

$$\frac{e :: \tau_0 \to \tau_1 \to \ldots \to \tau_n \qquad e \text{ is } \Gamma\text{-pure} \qquad \forall i.\ M'(\tau_i) = \tau_i}{\Gamma \vdash e \rightsquigarrow \langle \lambda t_0.\ \langle \lambda t_1.\ \cdots \langle \lambda t_{n-1}.\ e\ t_0\ t_1 \cdots t_{n-1}\rangle \cdots\rangle\rangle}\ \textsc{Pure}$$

$$\frac{\Gamma[x \mapsto \langle x'_{\mathsf{m}}\rangle] \vdash t \rightsquigarrow t_{\mathsf{m}}}{\Gamma \vdash (\lambda x :: \tau.\ t) \rightsquigarrow \langle \lambda x'_{\mathsf{m}} :: M'(\tau).\ t_{\mathsf{m}}\rangle}\ \lambda \qquad \frac{\Gamma \vdash e \rightsquigarrow e_{\mathsf{m}} \qquad \Gamma \vdash x \rightsquigarrow x_{\mathsf{m}}}{\Gamma \vdash (e\ x) \rightsquigarrow (e_{\mathsf{m}} \bullet x_{\mathsf{m}})}\ \textsc{App}$$

$$\frac{g \in \mathsf{dom}\,\Gamma}{\Gamma \vdash g \rightsquigarrow \Gamma(g)}\ \Gamma$$

The rules are ordered from highest to lowest priority.

An additional rule specifically treats case-expressions:

$$\frac{\begin{array}{c} g \text{ is a case-combinator with } n \text{ branches} \\ \Gamma \vdash t_1 \rightsquigarrow_\eta t'_1 \qquad \ldots \qquad \Gamma \vdash t_n \rightsquigarrow_\eta t'_n \end{array}}{\Gamma \vdash g\ t_1\ \ldots\ t_n \rightsquigarrow \langle g\ t'_1\ \ldots\ t'_n\rangle}\ \textsc{Comb}$$

Here $\Gamma \vdash t \rightsquigarrow_\eta t'$ denotes that t is fully η-expanded to $\lambda x_1 \ldots x_k.\, s$ first, and if $\Gamma[x_1 \mapsto \langle x'_1\rangle, \ldots, x_k \mapsto \langle x'_k\rangle] \vdash s \rightsquigarrow s'$, then $\Gamma \vdash t \rightsquigarrow_\eta \lambda x'_1 \ldots x'_k.\, s'$. The value that is subject to the case analysis is another argument t_{n+1}, and the arguments to $t_1, \ldots, t_n$ are produced by the case-combinator.

As an example, consider the following unusual definition of the Fibonacci sequence:

$$\mathit{fib}\,n = 1 + \mathit{sum}\,((\lambda f.\ \mathit{map}\,f\,[0..n-2])\,\mathit{fib})\,.$$

Its right-hand side can be transformed via the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\mathit{map} \in \mathsf{dom}\,\Gamma'}\,\Gamma \quad \cfrac{}{f \in \mathsf{dom}\,\Gamma'}\,\Gamma}{\Gamma' \vdash \mathit{map} \rightsquigarrow \mathit{map}_{\mathsf{m}}} \quad \cfrac{}{\Gamma' \vdash f \rightsquigarrow \langle f'_{\mathsf{m}}\rangle}\,\Gamma \quad \cfrac{}{[0..n-2] \rightsquigarrow \ldots}\,\diamond}{\langle \lambda f'_{\mathsf{m}}.\ \mathit{map}_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}}\rangle \bullet (\diamond)\rangle}\,\lambda \quad \cfrac{\mathit{fib}}{\langle \mathit{fib}'_{\mathsf{m}}\rangle}\,\Gamma}{\ldots}}{\Gamma \vdash \mathit{fib}\,n \rightsquigarrow \langle \lambda x.\ \langle 1 + x\rangle \bullet (\langle \lambda \mathit{xs}.\ \langle \mathit{sum}\,\mathit{xs}\rangle\rangle \bullet (\langle \lambda f'_{\mathsf{m}}.\ \mathit{map}_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}}\rangle \bullet (\ldots)\rangle \bullet \langle \mathit{fib}'_{\mathsf{m}}\rangle))\rangle}$$

[1] If f has type $\tau_1 \to \cdots \to \tau_n$ and x has type τ, the variables f'_{m} and x_{m} are assumed to have types $M'(\tau_1) \to \cdots \to M'(\tau_n)$ and $M(\tau)$, respectively. For a term $x :: \tau$ that satisfies $M'(\tau) = \tau$, x and x'_{m} are used interchangeably.

$$\dfrac{\dfrac{([..])\,0}{\langle \lambda r.\ \langle [0..r] \rangle \rangle}\ \textsc{Pure} \quad \dfrac{(-)}{\langle \lambda x.\ \langle \lambda y.\ \langle x - y \rangle \rangle \rangle}\ \textsc{Pure} \quad \dfrac{n}{\langle n'_{\mathsf{m}} \rangle}\ \Gamma \quad \dfrac{2}{\langle 2 \rangle}\ \textsc{Pure}}{[0..n-2] \rightsquigarrow \langle \lambda r.\ \langle [0..r] \rangle \rangle_\bullet (\langle \lambda x.\ \langle \lambda y.\ \langle x - y \rangle \rangle \rangle_\bullet \langle n'_{\mathsf{m}} \rangle_\bullet \langle 2 \rangle)}\ =: \diamond$$

where $\Gamma' = \Gamma[f \mapsto \langle f'_{\mathsf{m}} \rangle]$ and $\Gamma = [\mathit{fib} \mapsto \langle \mathit{fib}'_{\mathsf{m}} \rangle, n \mapsto \langle n'_{\mathsf{m}} \rangle]$. Parts of the derivation tree have been elided, and the left-hand side and Γ have been left out where they are clear from the context. A double line represents multiple applications of the App-rule. Note that the term $\lambda f.\ \mathit{map}\ f\ [0..n-2]$ falls through the Pure-rule before being processed by the λ-rule. This is because the bound variable f has the function type $\tau = \mathit{int} \to \mathit{int}$, which means $M'(\tau) \neq \tau$.

2.3 Reasoning with Parametricity

We want to use *relational parametricity* [19, 22] to prove the correspondence between a program and its memoized version. A memory m is said to be *consistent* with a pure function $f :: 'a \to 'r$, if it only memoizes actual values of f: if m maps a to r, then $r = f\ a$. We will use a relation $\Downarrow_R\ v\ s$ to assert that, given a consistent state m, $\mathit{run\_state}\ s\ m$ will produce a consistent state and a computation result v' with $R\ v\ v'$. Formally, $\Downarrow$ is defined with respect to the function f that we want to memoize, and an invariant $\mathit{inv}_{\mathsf{m}}$ on states:

$$\begin{aligned} \Downarrow_R\ v\ s\ =\ & \forall m.\ \mathit{cmem}\ m \wedge \mathit{inv}_{\mathsf{m}}\ m \longrightarrow \\ & (\mathsf{case}\ \mathit{run\_state}\ s\ m\ \mathsf{of}\ (v',\ m') \Rightarrow R\ v\ v' \wedge \mathit{cmem}\ m' \wedge \mathit{inv}_{\mathsf{m}}\ m') \end{aligned}$$

where $\mathit{cmem}\ m$ expresses that m is consistent with f and $\mathit{inv}_{\mathsf{m}}\ m$ expresses that m correctly represents a memory. Using the function relator

$$R \dashrightarrow S = \lambda f\ g.\ \forall x\ y.\ R\ x\ y \longrightarrow S\ (f\ x)\ (g\ y)\ ,$$

one can state the parametricity theorems for our fundamental monad combinators as (the relations can be understood as types):

$$\begin{aligned} (R \dashrightarrow \Downarrow_R)\ (\lambda x.\ x)\ \mathit{return} \\ (\Downarrow_R \dashrightarrow (R \dashrightarrow \Downarrow_S) \dashrightarrow \Downarrow_S)\ (\lambda v\ g.\ g\ v)\ (\gg\!=) \\ (\Downarrow_{(R \dashrightarrow \Downarrow_S)} \dashrightarrow \Downarrow_R \dashrightarrow \Downarrow_S)\ (\lambda\ g\ x.\ g\ x)\ (\bullet)\ . \end{aligned}$$

To prove the parametricity theorem, e.g. for map and $\mathit{map}'_{\mathsf{m}}$, one needs only the first and third property, and the parametricity theorems for all previously monadified constants that appear in map and $\mathit{map}'_{\mathsf{m}}$.

To prove the correspondence theorem for a monadification result, we use induction (following the recursion structure of the monadified function) together with parametricity reasoning.

Automating this induction proof is non-trivial. The reason is that the function definition command uses an elaborate extraction procedure based on congruence rules [11, 20] to extract recursive function calls and a surrounding context, which is used to prove termination of the function and to prove the induction

theorem. This may lead to complex (and necessary) assumptions in the induction hypotheses that specify for which sets of function arguments the hypotheses are valid. The challenge is to integrate the specific format of these assumptions with parametricity reasoning.

To combat this problem, we opt for a more specialized approach that performs an induction proof by exploiting parametricity but that goes beyond the infrastructure that is provided by Isabelle's integrated parametricity reasoning facility [12]. The main difference is that we use special variants of parametricity theorems that resemble the structure of the congruence theorems used by the function definition command. These produce the right pre-conditions to discharge the induction hypotheses.

Consider the *fib* function defined at the end of section 2.2. Both its termination and its generated induction rule are based on the information that *fib* n can call *fib* x only if x is between 0 and $n-2$. This information is extracted with the help of this congruence rule:

$$\frac{xs = ys \qquad \forall x.\ x \in set\ ys \longrightarrow f\ x = g\ x}{(map\ f\ xs) = (map\ g\ ys)}\ map\_cong$$

Similarly, for the monadified version fib'_{m}, its recursive calls are extracted by a pre-registered congruence rule:

$$\frac{xs = ys \qquad \forall x.\ x \in set\ ys \longrightarrow f'_{\mathsf{m}}\ x = g'_{\mathsf{m}}\ x}{(map_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}}\rangle \bullet \langle xs\rangle) = (map_{\mathsf{m}} \bullet \langle g'_{\mathsf{m}}\rangle \bullet \langle ys\rangle)}\ map_{\mathsf{m}}\_cong$$

After initiating the induction proof of the correspondence theorem with our tool, we are left with a goal that grants us the induction hypothesis

$$\forall x.\ x \in set\ [0, \ldots, n-2] \longrightarrow \Downarrow_{=}\ (fib\ x)\ (fib'_{\mathsf{m}}\ x)$$

and asks us to prove

$$\Downarrow_{list\_all2\ (=)}\ \ (map\ fib\ [0, \ldots, n-2])\ \ (map_{\mathsf{m}} \bullet \langle fib'_{\mathsf{m}}\rangle \bullet \langle [0, \ldots, n-2]\rangle)$$

where *list_all2* S compares two lists of equal length elementwise by relation S. To solve this goal, our tool will apply another pre-registered parametricity theorem for *map* and map_{m} (which is derived from map_{m}_cong following a canonical pattern):

$$\frac{xs = ys \qquad \forall x.\ x \in set\ ys \longrightarrow \Downarrow_S\ (f\ x)\ (f'_{\mathsf{m}}\ x)}{\Downarrow_{list\_all2\ S}\ (map\ f\ xs)\ (map_{\mathsf{m}} \bullet \langle f'_{\mathsf{m}}\rangle \bullet \langle ys\rangle)}\ map\_map_{\mathsf{m}}$$

It generates the same pre-condition as the congruence rule and yields a goal that exatcly matches the aforementioned induction hypothesis.

2.4 Termination

When monadifying a recursive function f that was defined with the function definition command [11], we need to replay the termination proof of f to define

f'_{m}. The termination proof—whether automatic or manual—relies on exhibiting a well-founded relation that is consistent with the recursive function calls of f. To capture the latter notion, the function definition command defines a relation *f_rel* between values in the domain of f. The idea for replaying the termination proof is that f and f'_{m} share the same domain and the same structure of recursive function calls. Thus, one tries to prove $f\_rel = f'_{\mathsf{m}}\_rel$, and if this succeeds, the termination relation for f is also compatible with the one for f'_{m}, yielding termination of f'_{m}.

However, the structure of *f_rel* is sometimes too dissimilar to f'_{m}_rel, and thus an automated proof of the equality fails. The main reason for that is that monadification can reorder the control flow and thus can alter the order in which the function definition commands encounters the recursive function calls when analyzing f'_{m}_rel. Moreover, sometimes a congruence rule is unnecessarily used while defining f, causing our tool to fail if a corresponding parametric version has not been registered with our tool. In such cases, we try to fall back to the automated termination prover that is provided by the function definition command.

2.5 Technical Limitations

While the monadification procedure that was presented in the previous sections is designed to run automatically, it is not universally applicable to any Isabelle/HOL function without previous setup. This encompasses the following limiations:

- As outlined above, higher-order combinators such as *map* generally need to be pre-registered together with their corresponding congruence and parametricity theorems.
- Just like Isabelle's function definition command, our tool relies on a context analysis for recursive calls. If we define (note the *id*)

  $$\mathit{fib}\, n = 1 + \mathit{sum}\, (\mathit{id}\, (\lambda f.\ \mathit{map}\, f\, [0..n-2])\, \mathit{fib})$$

  it becomes impossible to prove termination with the function definition command because the information that recursive calls happen only on values between 0 and $n-2$ is lost, and similarly our parametricity reasoner fails.
- Currently, our parametricity reasoner can only prove goals of the form

  $$(R \dashrightarrow S)(\lambda x.\ f\ x)(\lambda y.\ g\ y)$$

  if Isabelle's built-in parametricity reasoner can automatically show $R = (=)$. We plan to relax this limitation in the future.

Nevertheless, our tool works fully automatically for our case studies consisting of functions on lists and numbers that involve different higher-order combinators and non-trivial termination proofs.

2.6 Memoization

Compared to *monadification*, *memoization* of a program simply differs by replacing $=$ in each defining equation by $=_{\mathsf{m}}$ of type

$$('a \to ('m, 'r)\ state) \times 'a \to ('m, 'r)\ state \to bool\ .$$

The memoized version of a function of type $'a \to 'r$ then is of type $'a \to M'('r)$ where $'a$ should not contain function types. This seems to work only for functions with exactly one argument but our tool will automatically uncurry the function subject to memoization whenever necessary.

Concerning the memory type $'m$, we merely assume that it comes with two functions with the obvious intended meaning:

$$\begin{aligned} lookup &:: 'a \to ('m,\ 'r\ option)\ state \\ update &:: 'a \to 'r \to ('m, unit)\ state \end{aligned}$$

We use a memoizing operation *retrieve_or_run* to define $=_{\mathsf{m}}$:

$$\begin{aligned} \big((f'_{\mathsf{m}}, x) =_{\mathsf{m}} t\big) &= \big(f'_{\mathsf{m}}\ x = retrieve\_or\_run\ x\ (\lambda\_.\ t)\big) \\ retrieve\_or\_run\ x\ t' &= lookup\ x \gg\!= \big(\lambda r.\ \mathsf{case}\ r\ \mathsf{of} \\ &\qquad Some\ v \Rightarrow \langle v\rangle \\ &\quad \mid None \quad \Rightarrow t'\ (\,) \gg\!= (\lambda v.\ update\ x\ v \gg\!= \lambda\_.\ \langle v\rangle)\big)\ . \end{aligned}$$

Note that it is vital to wrap the additional λ-abstraction around t: otherwise call-by-value evaluation would build up a monadic expression that eagerly follows the full recursive branching of the original computation before any memoization is applied.

In order to specify the behavior of *lookup* and *update* we define an abstraction function $map\_of :: 'm \to 'a \to 'r\ option$ that turns a memory into a function:

$$map\_of\ heap\ k\ =\ fst\ (run\_state\ (lookup\ k)\ heap)\ .$$

To guarantee that *retrieve_or_run* always produces a consistent memory, *lookup* k should never add to the mapping, and *update* $k\ v$ should add at most the mapping $k \mapsto v$. (We will exploit the permissiveness of this specification in Section 2.9.) Formally, for all m with $inv_{\mathsf{m}}\ m$:

$$\begin{aligned} map\_of\ (snd\ (run\_state\ (lookup\ k)\ m)) &\subseteq_m map\_of\ m \\ map\_of\ (snd\ (run\_state\ (update\ k\ v)\ m)) &\subseteq_m (map\_of\ m)(k \mapsto v) \end{aligned}$$

where $(m_1 \subseteq_m m_2) \iff (\forall a \in dom\ m_1.\ m_1\ a = m_2\ a)$. Additionally, inv_{m} is required to be invariant under *lookup* and *update*. This allows us to prove correctness of *retrieve_or_run*:

$$\Downarrow_=\ (f\ x)\ s \longrightarrow \Downarrow_=\ (f\ x)\ (retrieve\_or\_run\ x\ s)\ .$$

Given that this is not a parametricity theorem, our method to inductively prove parametricity theorems for memoized functions needs to treat equations defined via $=_{\mathsf{m}}$ specially before parametric reasoning can be initiated.

From the correctness of *retrieve_or_run* and the correspondence theorem for f'_{m} we can derive correctness of f'_{m}:

$$\Downarrow_{=}\ (f\ x)\ (f'_{\mathsf{m}}\ x)\ .$$

As a corollary, we obtain:

$$\mathit{run\_state}\ (f'_{\mathsf{m}}\ x)\ \mathit{empty} = (v,\ m) \longrightarrow f\ x = v \wedge \mathit{cmem}\ m\ .$$

A simple instantiation of our memory interface can be given with the help of the standard implementation of mappings via red-black trees in Isabelle/HOL.

2.7 Imperative Memoization

This section outlines how our approach to monadification and memoization can be extended from a purely functional to an imperative memory implementation. Imperative HOL [4] is a framework for specifying and reasoning about imperative programs in Isabelle/HOL. It provides a *heap monad*

$$\textbf{datatype}\ 'a\ \mathit{Heap} = \mathit{Heap}\ (\mathit{execute} : \mathit{heap} \to ('a \times \mathit{heap})\ \mathit{option})$$

in which imperative programs can be expressed. The definition shows that the heap monad merely encapsulates a state monad (specialized to heaps) in an *option* monad to indicate failure. Our approach is simple: assuming that none of the operations in the memoized program fail (failures could only arise from *lookup* or *update*), the heap monad is equivalent to a state monad. This can be stated formally, where inv_h is a heap invariant:

$$\begin{aligned}&\Downarrow^h_R\ f_m\ f_h\ =\ \forall\ \mathit{heap}.\ \mathit{inv}_h\ \mathit{heap} \longrightarrow\\ &\quad(\mathsf{case}\ \mathit{run\_state}\ f_m\ \mathit{heap}\ \mathsf{of}\ (v_1, \mathit{heap}_1) \Rightarrow \mathsf{case}\ \mathit{execute}\ f_h\ \mathit{heap}\ \mathsf{of}\\ &\qquad \mathit{Some}\ (v_2, \mathit{heap}_2) \Rightarrow R\ v_1\ v_2 \wedge \mathit{heap}_1 = \mathit{heap}_2 \wedge \mathit{inv}_h\ \mathit{heap}_2\\ &\quad \mid \mathit{None} \Rightarrow \mathit{False})\end{aligned}$$

One could now be tempted to combine $\Downarrow$ and $\Downarrow^h$ into a relation between pure values and the heap monad by defining $\Downarrow'$ as a composition of the two:

$$\Downarrow'_R\ \ =\ \ \Downarrow_R \circ\circ \Downarrow^h_=$$

where $\circ\circ$ is the composition of binary relations. However, this would prohibit proving the analogue of the parametricity theorem for $\gg=$. The reason is that $\Downarrow'$ would demand too strong a notion of non-failure: computations are never allowed to fail, no matter whether we start the computation with a consistent state or not. Instead we use a weaker notion (analogous to $\Downarrow$)

$$\begin{aligned}&\Downarrow'_R\ v\ f_{\mathsf{h}}\ =\ \forall \mathit{heap}.\ \mathit{inv}_{\mathsf{m}}\ \mathit{heap}\ \wedge \mathit{inv}_{\mathsf{h}}\ \mathit{heap}\ \wedge \mathit{cmem}\ \mathit{heap}\ \longrightarrow\\ &\quad(\mathsf{case}\ \mathit{execute}\ f_{\mathsf{h}}\ \mathit{heap}\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{False}\\ &\qquad \mid \mathit{Some}\ (v',\ \mathit{heap}') \Rightarrow \mathit{inv}_{\mathsf{m}}\ \mathit{heap}' \wedge \mathit{inv}_{\mathsf{h}}\ \mathit{heap}' \wedge R\ v\ v' \wedge \mathit{cmem}\ \mathit{heap}')\end{aligned}$$

where inv_{m} and inv_{h} correspond to $\Downarrow$ and $\Downarrow^h$, respectively. The advantage is that one can prove

$$(\Downarrow_R \circ\!\circ \Downarrow^h_=)\ v\ f_{\mathsf{h}} \implies \Downarrow'_R\ v\ f_{\mathsf{h}}\ ,$$

to exploit compositionality where necessary, while still obtaining the analogous theorems for the elementary monad combinators (though not through reasoning via compositionality for $\gg\!=$). Using $\Downarrow'_R$ instead of $\Downarrow_R$, one can now use the same infrastructure for monadification and parametricity proofs to achieve imperative memoization.

2.8 Bottom-up Computation

In a classic imperative setting, dynamic programming algorithms are usually not expressed as recursive programs with memoization but rather as a computation that incrementally fills a table of memoized values according to some iteration strategy (typically in a bottom-up manner), using the recurrences to compute new values. The increased control over the computation order allows one to reduce the size of the memory drastically for some dynamic programming algorithms—examples of these can be found below. We propose a combination of two simple techniques to accomplish a similar behaviour and memory efficiency within our framework. The first, which is described in this section, is a notion of iterators for computations in the state monad that allows one to freely specify the computation order of a dynamic program. The second is to exploit our liberal interface for memories to use implementations that store only part of the previously seen computation results (to be exemplified in the next section).

Our interface for iterators consists of two functions $cnt :: 'a \to bool$ and $nxt :: 'a \to 'a$ that indicate whether the iterator can produce any more elements and yield the next element, respectively. We can use these to iterate a computation in the state monad:

$$\begin{aligned} &iter\_state\ f\ =\ wfrec\ \{(nxt\ x,\ x)\mid cnt\ x\}\\ &\quad(\lambda rec\ x.\ \mathsf{if}\ cnt\ x\ \mathsf{then}\ f\ x \gg\!= (\lambda\_.\ rec\ (nxt\ x))\ \mathsf{else}\ \langle(\,)\rangle) \end{aligned}$$

where *wfrec* takes the well-founded termination relation as its first argument. Given a size function on the iterator value, we can prove termination if

$$finite\ \{x \mid cnt\ x\}\ \text{ and }\ \forall\, x.\ cnt\ x \longrightarrow size\ x < size\ (nxt\ x)\ .$$

Provided that a given iteration strategy terminates in this sense, we can use it to compute the value of a memoized function:

$$(= \dashrightarrow \Downarrow_R)\ g\ f\ \longrightarrow\ \Downarrow_R\ (g\ x)\ (iter\_state\ cnt\ nxt\ f\ x \gg\!= (\lambda\_.\ f\ x))\ .$$

As an example, a terminating iterator that builds up a table of n rows and m columns in a row-by-row, left-to-right order can be specified as:

$$\begin{aligned} size\ (x,y) &= x * (m+1) + y\\ cnt\ (x,y) &= x \le n \wedge y \le m\\ nxt\ (x,y) &= \mathsf{if}\ y < m\ \mathsf{then}\ (x, y+1)\ \mathsf{else}\ (x+1, 0) \end{aligned}$$

If the recursion pattern of f is consistent with *nxt*, the stack depth of the iterative version is at most one because every recursive call is already memoized.

2.9 Memory Implementations

To achieve a space-efficient implementation for the Minimum-Edit Distance problem or the Bellman-Ford algorithm, one needs to complement the bottom-up computation strategy from the last section with a memory that stores only the last two rows. We will showcase how such a memory can be implemented generically within our framework, and how to exploit compositionality to get an analogous imperative implementation without repeating the correctness proof.

Abstractly, we will implement a mapping $'k \to 'v\ option$ and split up $'k$ using two key functions $key_1 :: 'k \to 'k_2$ and $key_2 :: 'k \to 'k_1$. We demand that together, they are injective:

$$\forall k\ k'.\ key_1\ k = key_1\ k' \wedge key_2\ k = key_2\ k' \longrightarrow k = k'\ .$$

For a rectangular memory, for instance, key_1 and key_2 could map a key to its row and column index. We use two pairs of lookup and update functions, (l_1, u_1) and (l_2, u_2) to implement the memory for the two rows. We also store the row keys k_1 and k_2 that the currently stored values correspond to in the memory.

For the verification it is crucial that we have previously introduced a memory invariant. The invariant states that k_1 and k_2 are different, and that the first and second row only store key-value pairs that correspond to k_1 and k_2, respectively. The main additional insight that is used in the correctness proof for this memory implementation is the following monotonicity lemma, where $\cup_m$ denotes map union:

$$\begin{aligned}&(m_1 \cup_m m_2) \subseteq_m (m_1' \cup_m m_2')\\ &\text{if } m_1 \subseteq_m m_1',\ m_2 \subseteq_m m_2',\ \text{and } dom\ m_1 \cap dom\ m_2' = \{\}\ .\end{aligned}$$

We now extend this formalization towards an imperative implementation that stores the two rows as arrays. To this end, assume we are also given a function $idx\_of :: 'k_2 \to nat$ with

$$\begin{aligned}&mem\_update\ k\ v = (\mathsf{let}\ i = idx\_of\ f\ k\ \mathsf{in}\\ &\quad \mathsf{if}\ i < size\ \mathsf{then}\ (Array.upd\ i\ (Some\ v)\ mem \gg\!= (\lambda\ \_.\ return\ ()))\\ &\quad \mathsf{else}\ return\ ()\end{aligned}$$

To verify this implementation, we wrap lookup, update, and move in

$$state\_of\ s\ =\ State\ (\lambda\ heap.\ the\ (execute\ s\ heap))$$

where $the\ (Some\ x) = x$, and prove that these correctly implement the interface for the previous implementation in the state monad. As the second step, one relates—via parametricity reasoning—this implementation with an implementation in the heap monad, where lookup, update and move are used without the $state\_of$ wrapper: we can prove $\Downarrow^h_= (state\_of\ m)\ m$ if m never fails and preserves the memory invariant.

3 Examples

This section presents five representative but quite different examples of dynamic programming. We have also applied the tool to further examples that are not explained here, for instance the optimal binary search tree problem [18] and the Viterbi algorithm [23]. For the first example, Bellman-Ford, we start with a recursive function, prove its correctness and refine it to an imperative memoized algorithm with the help of the automation described above. Because the refinement steps are automatic and the same for all examples, they are not shown for the other examples.

The examples below employ lists: $x \cdot xs$ is the list with head x and tail xs; xs @ ys is the concatenation of the lists xs and ys; $xs\,!\,i$ is the ith element of xs; $[i..j]$ is the list of integers from i to j, and similarly for the set $\{i..j\}$; $slice\ xs\ i\ j$ is the sublist of xs from index i (starting with 0) to (but excluding) index j.

For the verification of the Knapsack problem and the Bellman-Ford algorithm, we followed Kleinberg and Tardos [10]. In both cases, the crucial part of the correctness argument involves a recurrence of the form

$$OPT\ (Suc\ n)\ t_1\ \ldots\ t_k = \Pi\{u_1, \ldots, u_m\}$$

where each of the u_i involve terms of the form $OPT\ n$ and $\Pi \in \{Min, Max\}$. We prove this equality by proving two inequalities ($\leq$, $\geq$). The easier direction is the one where we just need to show that the left-hand side covers all the solutions that are covered by the right-hand side. This direction is not explicitly covered in the proof by Kleinberg and Tardos. For the other direction, we first prove that the unique minimum or maximum exist and then analyze the solution that computes the minimum or maximum, directly following the same kind of argument as Kleinberg and Tardos.

3.1 Bellman-Ford Algorithm

The Bellman-Ford Algorithm solves the *single-destination shortest path problem* (and the single-source shortest path problem by reversing the edges): given nodes $1, \ldots, n$, a sink $t \in \{1, \ldots, n\}$, and edge weights $W :: nat \to nat \to int$, we have to compute for each source $v \in \{1, \ldots, n\}$ the minimum weight of any path from v to t. The main idea of the algorithm is to consider paths in the order of increasing *path length*. Thus we define $OPT\ i\ v$ as the weight of the shortest path leading from v to t, and using at most i edges:

$$\begin{aligned} OPT\ i\ v = Min\ (\{&\mathsf{if}\ t = v\ \mathsf{then}\ 0\ \mathsf{else}\ \infty\} \cup \\ &\{weight\ (v \cdot xs) \mid length\ xs + 1 \leq i \wedge set\ xs \subseteq \{0..n\}\})\ . \end{aligned}$$

If $OPT\ (n+1)\ s = OPT\ n\ s$ for all $s \in \{1, \ldots, n\}$, then there is no cycle of negative weight (from which t can be reached), and $OPT\ n$ represents shortest path lengths. Otherwise, we know that there is a cycle of negative weight.

Following Kleinberg and Tardos, we prove

$$OPT\ (Suc\ i)\ v = min\ (OPT\ i\ v)\ (Min\ \{OPT\ i\ w + W\ v\ w \mid w.\ w \leq n\})\ ,$$

yielding a recursive solution (replacing sets by lists):

$$\begin{aligned} &BF\ 0\ j &&= (\mathsf{if}\ t = j\ \mathsf{then}\ 0\ \mathsf{else}\ \infty) \\ &BF\ (Suc\ k)\ j &&= min\_list\ (BF\ k,\ j \cdot [W\ j\ i + BF\ k,\ i\ .\ i \leftarrow [0..n]])\ . \end{aligned}$$

Applying our tool for memoization, we get:

$$\begin{aligned} &BF_m{}'\ 0\ j &&=_{\mathsf{m}} if_{\mathsf{m}}\ \langle t = j\rangle\ \langle 0\rangle\ \langle\infty\rangle \\ &BF_m{}'\ (Suc\ k)\ j &&=_{\mathsf{m}} \langle\lambda xs.\ \langle min\_list\ xs\rangle\rangle \bullet (\langle\lambda x.\ \langle\lambda xs.\ \langle x \cdot xs\rangle\rangle\rangle \bullet BF_m{}'\ k\ j \bullet \\ & && \quad (map_{\mathsf{m}} \bullet \langle\lambda i.\ \langle\lambda x.\ \langle W\ j\ i\ +\ x\rangle\rangle \bullet BF_m{}'\ k\ i\rangle \bullet \langle[0..n]\rangle))\ . \end{aligned}$$

Using the technique described in section 2.8, we fill the table in the order $(0,0),(0,1),\ldots,(n,0),\ldots,(n,n)$. The pairwise memory implementation from section 2.9 is used to only store two rows corresponding to the first part of the pair, which are in turn indexed by the second one. Together, this yields a program that can compute the length of the shortest path in $O(n)$ space. The final correctness theorem for this implementation is (with explicit context parameters n and W):

$$\begin{aligned} &BF\ n\ W\ t\ i\ j = fst\ (run\_state \\ &\quad (iter\_BF\ n\ W\ t\ (i,\ j) \gg\!= (\lambda\ \_.\ BF_m{}'\ n\ W\ t\ i\ j))\ Mapping.empty)\ . \end{aligned}$$

Isabelle can be instructed to use this equation when generating code for BF. Thus the efficient implementation becomes completely transparent for the user.

Lastly, we can choose how to implement the parameter for the edge weights W. A common graph representation are adjacency lists of type $(nat \times int)\ list\ list$ that contain for each node v an association list of pairs of a neighbouring node and the corresponding edge weight. To obtain an efficient implementation, the outer list can be realized with Isabelle's immutable arrays. They come with a function $IArray$ that maps $'a\ list$ to an immutable array and with the infix !! array subscript function. Thus we can transform a list into an immutable array first and then run the Bellman-Ford algorithm:

$$\begin{aligned} &BF\_ia\ n\ W\ t\ i\ j = (\mathsf{let}\ W' = graph\_of\ (IArray\ W)\ \mathsf{in}\ fst\ (run\_state \\ &\qquad\qquad (iter\_BF\ n\ W'\ t\ (i,\ j) \gg\!= (\lambda\ \_.\ BF_m{}'\ n\ W'\ t\ i\ j)) \\ &\qquad\qquad Mapping.empty)) \\ &graph\_of\ a\ i\ j\ \ = \mathsf{case}\ find\ (\lambda p.\ fst\ p = j)\ (a\ !!\ i)\ \mathsf{of} \\ &\qquad\qquad None \Rightarrow \infty \mid Some\ x \Rightarrow snd\ x\ . \end{aligned}$$

Note that the defining equation for $BF_h{}'$ looks exactly the same as for $BF_m{}'$ but for different underlying constants for the heap monad. For imperative memoization, the final theorems for BF or BF_ia would just differ in that run_state would be replaced by $execute$ and the initial memory would be replaced by a correctly initialized empty heap memory.

3.2 Knapsack Problem

In the Knapsack Problem, we are given n items $1,\ldots,n$, a weight assignment $w :: nat \to nat$, and a value assignment $v :: nat \to nat$. Given a Knapsack,

which can carry at most weight W, the task is to compute a selection of items that fits in the Knapsack and maximizes the total value. Thus we define:

$$OPT\ n\ W = Max \left\{ \sum_{i \in S} v\ i \;\middle|\; S \subseteq \{1..n\} \wedge \sum_{i \in S} w\ i\ \leq\ W \right\} .$$

Again following Kleinberg and Tardos, we prove:

$$\begin{aligned} &OPT\ (Suc\ i)\ W = (\mathsf{if}\ W < w\ (Suc\ i)\ \mathsf{then}\ OPT\ i\ W \\ &\quad \mathsf{else}\ max\ (v\ (Suc\ i) + OPT\ i\ (W - w\ (Suc\ i)))\ (OPT\ i\ W)) . \end{aligned}$$

This directly yields the following recursive solution:

$$\begin{aligned} &knapsack\ 0\ W\ =\ 0 \\ &knapsack\ (Suc\ i)\ W\ =\ (\mathsf{if}\ W < w\ (Suc\ i)\ \mathsf{then}\ knapsack\ i\ W \\ &\quad \mathsf{else}\ max\ (knapsack\ i\ W)\ (v\ (Suc\ i) + knapsack\ i\ (W - w\ (Suc\ i)))) . \end{aligned}$$

Like Bellman-Ford, the algorithm can be memoized using a bottom-up computation and a memory, which stores only the last two rows. However, the algorithm's running time and space consumption are still exponential in the input size, assuming a binary encoding of W.

3.3 A Counting Problem

A variant of Project Euler problem #114 [2] was posed in the 2018 edition of the "VerifyThis" competition [3] [14]. We consider a row consisting of n tiles, which can be either red or black, and we impose the condition that red tiles only come in blocks of three consecutive tiles. We are asked to compute $count(n)$, the number of valid rows of size n. This is an example of counting problems that can be solved with memoization.

Besides the base cases $count(0) = count(1) = count(2) = 1$, and $count(3) = 2$, one gets the following recursion:

$$count(n) = count(n-1) + 1 + \sum_{i=3}^{n-1} count(n-i-1)\ \ \text{if}\ \ n > 3 .$$

These equations directly yield a recursive functional solution, which can be memoized as described for the examples above. The reasoning to prove the main recursion, however, is different. We define

$$count(n) = card\ \{l \mid length\ l = n \wedge valid\ l\}$$

where *valid* is an inductively defined predicate describing a well-defined row. The reasoning trick is to prove the following case analysis on the validity of a single row

$$\begin{aligned} valid\ l \longleftrightarrow\ & l = [] \vee (l\ !\ 0 = B \wedge valid\ (tl\ l)) \vee \\ & length\ l \geq 3 \wedge (\forall i < length\ l.\ l\ !\ i = R) \vee \ldots . \end{aligned}$$

that is then used to split the defining set of $count(n)$ into *disjoint* subsets that correspond to the individual terms on the right-hand side of the recursion.

[2] `https://projecteuler.net/problem=114`

[3] `http://www.pm.inf.ethz.ch/research/verifythis.html`

3.4 The Cocke-Younger-Kasami Algorithm

Given a grammar in Chomsky normal form, the CYK algorithm computes the set of nonterminals that produce (yield) some input string. We model productions in Chomsky normal form as pairs (A,r) of a nonterminal A and a r.h.s. r that is either of the form $T\ a$, where a is a terminal (of type $'t$), or $NN\ B\ C$, where B and C are nonterminals (of type $'n$). Below, $P :: ('n,'t)\ prods$ is a list of productions. The *yield* of a nonterminal is defined inductively as a relation:

$$\frac{(A,\ T\ a) \in set\ P}{yield\ P\ A\ [a]} \qquad \frac{(A,\ NN\ B\ C) \in set\ P \qquad yield\ P\ B\ u \qquad yield\ P\ C\ v}{yield\ P\ A\ (u\ @\ v)}$$

A functional programmer will start out with an implementation $CYK :: ('n, 't)\ prods \to 't\ list \to 'n\ list$ of the CYK algorithm defined by recursion on lists and prove its correctness: $set\ (CYK\ P\ w) = \{N \mid yield\ P\ N\ w\}$. However, memoizing the list argument leads to an inefficient implementation. An efficient implementation can be obtained from a version of the CYK algorithm that indexes into the (constant) list and memoizes the index arguments. Our starting point is the following function *CYK_ix* where w is not of type $'a\ list$ but an indexing function of type $nat \to 't$. Isabelle supports list comprehension syntax:

$$\begin{array}{l}
CYK_ix :: ('n,\ 't)\ prods \to (nat \to 't) \to nat \to nat \to 'n\ list \\
CYK_ix\ P\ w\ i\ 0 = [] \\
CYK_ix\ P\ w\ i\ (Suc\ 0) = [A\ .\ (A,\ T\ a) \leftarrow P,\ a = w\ i] \\
CYK_ix\ P\ w\ i\ n = \\
[A.\ k \leftarrow [1..n{-}1],\ B \leftarrow CYK_ix\ P\ w\ i\ k,\ C \leftarrow CYK_ix\ P\ w\ (i{+}k)\ (n{-}k), \\
\quad (A,\ NN\ B'\ C') \leftarrow P,\ B' = B,\ C' = C]
\end{array}$$

The correctness theorem (proved by induction) explains the meaning of the arguments i and n:

$$set\ (CYK\_ix\ P\ w\ i\ n) = \{N \mid yield\ P\ N\ (slice\ w\ i\ (i + n))\}$$

As for Bellman-Ford, we obtain an imperative memoized version $CYK\_ix'_{\mathsf{m}}$ and a correctness theorem that relates it to *CYK_ix* and parameter w is realized by an immutable array.

3.5 Minimum Edit Distance

The minimum edit distance between two lists xs and ys of type $'a\ list$ is the minimum cost of converting xs to ys by means of a sequence of the edit operations copy, replace, insert and delete:

datatype $'a\ ed = Copy \mid Repl\ 'a \mid Ins\ 'a \mid Del$

The cost of *Copy* is 0, all other operations have cost 1. Function *edit* defines how an $'a\ ed\ list$ transform one $'a\ list$ into another:

$$
\begin{array}{l}
\mathit{edit}\ (\mathit{Copy} \cdot \mathit{es})\ (x \cdot \mathit{xs}) = x \cdot \mathit{edit}\ \mathit{es}\ \mathit{xs} \\
\mathit{edit}\ (\mathit{Repl}\ a \cdot \mathit{es})\ (\_ \cdot \mathit{xs}) = a \cdot \mathit{edit}\ \mathit{es}\ \mathit{xs} \\
\mathit{edit}\ (\mathit{Ins}\ a \cdot \mathit{es})\ \mathit{xs} = a \cdot \mathit{edit}\ \mathit{es}\ \mathit{xs} \\
\mathit{edit}\ (\mathit{Del} \cdot \mathit{es})\ (\_ \cdot \mathit{xs}) = \mathit{edit}\ \mathit{es}\ \mathit{xs} \\
\mathit{edit}\ []\ \mathit{xs} = \mathit{xs}
\end{array}
$$

We have omitted the cases where the second list becomes empty before the first.

This time we start from two functions defined by recursion on lists:

$$
\begin{array}{l}
\mathit{min\_ed} :: {'a}\ \mathit{list} \to {'a}\ \mathit{list} \to \mathit{nat} \\
\mathit{min\_eds} :: {'a}\ \mathit{list} \to {'a}\ \mathit{list} \to {'a}\ \mathit{ed}\ \mathit{list}
\end{array}
$$

Function *min_ed* computes the minimum edit distance and $\mathit{min\_eds} :: {'a}\ \mathit{list} \to {'a}\ \mathit{list} \to {'a}\ \mathit{ed}\ \mathit{list}$ computes a list of edits with minimum cost. We omit their definitions. The relationship between them is trivial to prove: $\mathit{min\_ed}\ \mathit{xs}\ \mathit{ys} = \mathit{cost}\ (\mathit{min\_eds}\ \mathit{xs}\ \mathit{ys})$. Therefore the following easy correctness and minimality theorems about *min_eds* also imply correctness and minimality of *min_ed*:

$$\mathit{edit}\ (\mathit{min\_eds}\ \mathit{xs}\ \mathit{ys})\ \mathit{xs} = \mathit{ys} \qquad \mathit{cost}\ (\mathit{min\_eds}\ \mathit{xs}\ (\mathit{edit}\ \mathit{es}\ \mathit{xs})) \leq \mathit{cost}\ \mathit{es}$$

As for CYK, we define a function by recursion on indices

$$
\begin{array}{l}
\mathit{min\_ed\_ix} :: (\mathit{nat} \to {'a}) \to (\mathit{nat} \to {'a}) \to \mathit{nat} \to \mathit{nat} \to \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\
\mathit{min\_ed\_ix}\ \mathit{xs}\ \mathit{ys}\ m\ n\ i\ j = \\
(\mathsf{if}\ m \leq i\ \mathsf{then}\ \mathsf{if}\ n \leq j\ \mathsf{then}\ 0\ \mathsf{else}\ n - j \\
\ \mathsf{else\ if}\ n \leq j\ \mathsf{then}\ m - i \\
\qquad \mathsf{else}\ \mathit{min\_list} \\
\qquad\qquad [1 + \mathit{min\_ed\_ix}\ \mathit{xs}\ \mathit{ys}\ m\ n\ i\ (j+1), \\
\qquad\qquad\ 1 + \mathit{min\_ed\_ix}\ \mathit{xs}\ \mathit{ys}\ m\ n\ (i+1)\ j, \\
\qquad\qquad\ (\mathsf{if}\ \mathit{xs}\ i = \mathit{ys}\ j\ \mathsf{then}\ 0\ \mathsf{else}\ 1) + \\
\qquad\qquad\ \mathit{min\_ed\_ix}\ \mathit{xs}\ \mathit{ys}\ m\ n\ (i+1)\ (j+1)])
\end{array}
$$

and prove that it correctly refines *min_ed*: $\mathit{min\_ed\_ix}\ \mathit{xs}\ \mathit{ys}\ m\ n\ i\ j = \mathit{min\_ed}\ (\mathit{slice}\ \mathit{xs}\ i\ m)\ (\mathit{slice}\ \mathit{ys}\ j\ n)$. Although one can prove correctness of this indexed version directly, the route via the recursive functions on lists is simpler.

As before we obtain an imperative memoized version $\mathit{min\_ed\_ix}'_{\mathsf{m}}$ and a correctness theorem that relates it to *min_ed_ix*.

4 Future Work

We plan to expand our work in two major directions in the future. Firstly, we want to use our memoization tool to allow for other monads than the state and the heap monad. The main task here is to find monads that play well with our style of parametric reasoning. In simple monads such as reader or writer monads, the monadic operations do not interfere with the original computation, so they fit well in this framework. For the state monad, we can give correspondence proofs because we thread an invariant—values stored in the state are consistent

with the memoized functions—through our relations. For other monads, such as an IO monad, it is less clear what these invariants would look like. Moreover, our tool currently only adds monadic effects at recursive invocations of a function—for other monads one would certainly want to insert these in other places, too. This added flexibility would also allow us to save recursive function invocations in memoized functions: instead of performing the memoization at the equality sign, we could wrap memoization around each recursive invocation of the function. Furthermore, this would allow one to memoize repeated applications of non-recursive functions in the context of an enclosing function.

Our second goal is to integrate the memoization process with the Imperative Refinement Framework [13]. It allows stepwise refinement of functional programs and to replace functional by imperative data structures in a final refinement step. The main obstacle here is that the framework already comes with its own nondeterminism monad to facilitate refinement reasoning. This means that high-level programs are already stated in terms of this monad. We have started work to allow automated monadification of these programs by adding the state via a state transformer monad.

Acknowledgments

Tobias Nipkow is supported by DFG Koselleck grant NI 491/16-1. The authors would like to thank Andreas Lochbihler for a fruitful discussion on monadification.

References

1. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). LNCS, vol. 2277, pp. 24–40. Springer (2002)
2. Bortin, M.: A formalisation of the cocke-younger-kasami algorithm. Archive of Formal Proofs (2016), `http://isa-afp.org/entries/CYK.html`, Formal proof development
3. Braibant, T., Jourdan, J., Monniaux, D.: Implementing and reasoning about hash-consed data structures in Coq. J. Autom. Reasoning **53**(3), 271–304 (2014), `https://doi.org/10.1007/s10817-014-9306-0`
4. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics (TPHOLs 2008). pp. 134–149 (2008)
5. Erwig, M., Ren, D.: Monadification of functional programs. Science of Computer Programming **52**(1), 101 – 129 (2004), `http://www.sciencedirect.com/science/article/pii/S0167642304000486`, special Issue on Program Transformation
6. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 100–115. Springer (2013)
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Functional and Logic Programming (FLOPS 2010). LNCS, vol. 6009, pp. 103–117. Springer (2010)

8. Hatcliff, J., Danvy, O.: A generic account of continuation-passing styles. In: Conf. Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 458–471 (1994), http://doi.acm.org/10.1145/174675.178053
9. Itzhaky, S., Singh, R., Solar-Lezama, A., Yessenov, K., Lu, Y., Leiserson, C., Chowdhury, R.: Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In: Proc. 2016 ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications. pp. 145–164. OOPSLA 2016, ACM (2016), http://doi.acm.org/10.1145/2983990.2983993
10. Kleinberg, J.M., Tardos, É.: Algorithm Design. Addison-Wesley (2006)
11. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/doc/681651/document.pdf
12. Kuncar, O.: Types, Abstraction and Parametric Polymorphism in Higher-Order Logic. Ph.D. thesis, Technical University Munich, Germany (2016), http://nbn-resolving.de/urn:nbn:de:bvb:91-diss-20160408-1285267-1-5
13. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015, Proceedings. Lecture Notes in Computer Science, vol. 9236, pp. 253–269. Springer (2015)
14. Lammich, P., Wimmer, S.: VerifyThis 2018 — Polished Isabelle solutions. Archive of Formal Proofs (Apr 2018), http://isa-afp.org/entries/VerifyThis2018.html, Formal proof development
15. Michie, D.: Memo functions and machine learning. Nature **218**, 19–22 (1968)
16. Nipkow, T., Klein, G.: Concrete Semantics with Isabelle/HOL. Springer (2014), http://concrete-semantics.org
17. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
18. Nipkow, T., Somogyi, D.: Optimal binary search tree. Archive of Formal Proofs (2018), http://isa-afp.org/entries/Optimal_BST.html, Formal proof development
19. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
20. Slind, K.: Reasoning about terminating functional programs. Ph.D. thesis, Technical University Munich, Germany (1999), https://mediatum.ub.tum.de/node?id=601660
21. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting BDDs in Coq. In: He, J., Sato, M. (eds.) Advances in Computing Science - ASIAN 2000. LNCS, vol. 1961, pp. 162–181. Springer (2000)
22. Wadler, P.: Theorems for free! In: Proc. Fourth Int. Conf. Functional Programming Languages and Computer Architecture. pp. 347–359. FPCA '89, ACM (1989), http://doi.acm.org.eaccess.ub.tum.de/10.1145/99370.99404
23. Wimmer, S.: Hidden markov models. Archive of Formal Proofs (2018), http://isa-afp.org/entries/Hidden_Markov_Models.html, Formal proof development
24. Wimmer, S., Hu, S., Nipkow, T.: Monadification, memoization and dynamic programming. Archive of Formal Proofs (2018), http://isa-afp.org/entries/Monad_Memo_DP.html, Formal proof development