Memoizing a Monadic Mixin DSL

Pieter Wuille[1], Tom Schrijvers[2], Horst Samulowitz[3], Guido Tack[1], and Peter Stuckey[4]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Department of Applied Mathematics and Computer Science, UGent, Belgium
[3] IBM Research, USA
[4] National ICT Australia (NICTA) and University of Melbourne, Victoria, Australia

Abstract. Modular extensibility is a highly desirable property of a domain-specific language (DSL): the ability to add new features without affecting the implementation of existing features. Functional mixins (also known as open recursion) are very suitable for this purpose.
We study the use of mixins in Haskell for a modular DSL for search heuristics used in systematic solvers for combinatorial problems, that generate optimized C++ code from a high-level specification. We show how to apply memoization techniques to tackle performance issues and code explosion due to the high recursion inherent to the semantics of combinatorial search.
As such heuristics are conventionally implemented as highly entangled imperative algorithms, our Haskell mixins are monadic. Memoization of monadic components causes further complications for us to deal with.

1 Application domain

Search heuristics often make all the difference between effectively solving a combinatorial problem and utter failure. Heuristics enable a search algorithm to become efficient for a variety of reasons, e.g., incorporation of domain knowledge, or randomization to avoid heavy tailed runtimes. Hence, the ability to swiftly design search heuristics that are tailored towards a problem domain is essential to performance improvement. In other words, this calls for a high-level domain-specific language (DSL).

The tough technical challenge we face when designing a DSL for search heuristics, does not lie in designing a high-level syntax; several proposals have already been made (e.g., [10]). What is really problematic is to bridge the gap between a conceptually simple specification language (high-level and naturally compositional) and an efficient implementation (typically low-level, imperative and highly non-modular). This is indeed where existing approaches fail; they restrict the expressiveness of their DSL to face up to implementation limitations, or they raise errors when the user strays out of the implemented subset.

We overcome this challenge with a systematic approach that disentangles different primitive concepts into separate modular *mixin* components, each of which corresponds to a feature in the high-level DSL. The great advantage of

$s ::=$ prune
prunes the node
| base_search$(\ldots)$
label
| let(v, e, s)
introduce new global variable v with initial value e, then perform s
| assign(v, e)
assign e to variable v and succeed
| and$([s_1, s_2, \ldots, s_n])$
perform $s1$, on success start $s2$ otherwise fail, ...
| or$([s_1, s_2, \ldots, s_n])$
perform $s1$, on termination start $s2$, ...
| post(c, s)
perform s and post a constraint c at every node

Fig. 1. Syntax of Search Heuristics DSL

mixin components to provide a semantics for our DSL is its modular extensibility. We can add new features to the language by adding more mixin components. The cost of adding such a new component is small, because it does not require changes to the existing ones.

The application under consideration is heuristics for systematic tree search in the area of Constraint Programming (CP), but the same issues apply to other search-driven areas in the field of Artificial Intelligence (AI) and related areas such as Operations Research (OR). The goal is generating tight C++ code for doing search from our high-level DSL. The focus however lies in the combination of using Haskell combinators for expressing strategies, open recursion to allow modular extension and monads for allowing stateful behaviour to implement a code-generation system. Further on, we explain how to combine this with memoization to improve generation time as well as size of the generated code.

2 Brief DSL Overview

We provide the user with a high-level domain-specific language (DSL) for expressing search heuristics. For this DSL we use a concrete syntax, in the form of nested terms, that is compatible with the *annotation* language of MiniZinc [9], a popular language for modeling combinatorial problems.

The search specification implicitly defines a search tree whose leaves are solutions to the given problem. Our implementation parses a MiniZinc model, extracts the search specification expressed in our DSL and generates the corresponding low-level C++ code for navigating the search tree. The remainder of the MiniZinc model (expressing the actual combinatorial problem) is shipped to the Gecode library [7], a state-of-the-art finite domain constraint solver. The search code interacts with the solver at every node of the search tree to determine whether a solution or dead end has been reached, or whether to generate new child nodes for further exploration.

2.1 DSL Syntax

The DSL's *expression language* comprises the typical arithmetic and comparison operators and literals that require no further explanation. Notable though is the fact that it allows referring to the constraint variables and parameters of the constraint model.

The DSL's *search heuristics language* features a number of primitives, listed in the catalog of Fig. 1, in terms of which more complex heuristics can be defined. The catalog consists of both *basic* heuristics and *combinators*. The former define complete (albeit very basic) heuristics by themselves, while the latter alter the behavior of one or more other heuristics.

There are two basic heuristics: prune, which cuts the search tree below the current node, and the base search strategies, which implement the *labeling* (also known as *enumeration*) strategies. We do not elaborate on the base search here, because this has been studied extensively in the literature. While only a few basic heuristics exist, the DSL derives great expressive power from the infinite number of ways in which these basic heuristics can be composed by means of combinators.

The combinator $\mathsf{let}(v, e, s)$ introduces a new variable v, initialized to the value of expression e, in the sub-search s, while $\mathsf{assign}(v, e)$ assigns the value of e to v and succeeds. The and-sequential composition $\mathsf{and}([s_1, \ldots, s_n])$ runs s_1 and at every success leaf runs $\mathsf{and}([s_2, \ldots, s_n])$. In contrast, $\mathsf{or}([s_1, \ldots, s_n])$ first runs s_1 in full before restarting with $\mathsf{or}([s_2, \ldots, s_n])$.

Finally, the $\mathsf{post}(c, s)$ primitive provides access to the underlying constraint solver, posting a constraint c at every node during s. If s is omitted, it posts the constraint and immediately succeeds.

As an example, this is how branch-and-bound — a typical optimization heuristic — can be expressed in the DSL:

$$\boxed{\mathsf{let}(best, \mathsf{maxint}, \mathsf{post}(obj < best, \mathsf{and}([\mathsf{base\_search}(\ldots), \mathsf{assign}(best, obj)])))}$$

let introduces the variable *best*, post makes sure the constraint $obj < best$ is enforced at each node of the search tree spawned by base_search. Combining it with assign using and causes the *best* variable to be updated after finding solutions. Note that we refer to *obj*, the program variable being minimized.

3 Implementation

Starting from base searches and functions for combining them — as called by the parser — a C++ AST is generated. After a simplification step, a pretty printer is invoked to generate the actual source code. Both the initial parsing phase and pretty printer are trivial and not discussed here.

3.1 C++ Abstract Syntax Tree

Before we discuss the code generator, we need to define the target language, a C++ AST, which is partly given here:

$$
\begin{array}{llll}
\textbf{data}\ \mathit{Stmt} = & \mathit{Nop} & \mid & \mathit{Expr} := \mathit{Expr} \\
\mid & \mathit{IfThenElse}\ \mathit{Expr}\ \mathit{Stmt}\ \mathit{Stmt} & \mid & \mathit{Stmt}\,;\mathit{Stmt} \\
\mid & \mathit{Call}\ \mathit{String}\ [\mathit{Expr}] & \mid & \mathit{While}\ \mathit{Expr}\ \mathit{Stmt} \\
\mid & \ldots & &
\end{array}
$$

A number of convenient abbreviations facilitate building this AST, e.g.,

$$
\begin{array}{l}
(\fatsemi) = \mathit{liftM} \quad \circ\ (;) \\
\mathit{if}' = \mathit{liftM2} \circ \mathit{IfThenElse}
\end{array}
$$

3.2 The Combinator stack

Based on the output of the parser, a data structure is built that represents the search heuristic. The details of how this is represented will follow later, but in general, a value of type *Search* will be used. Basic heuristics result immediately in a *Search*, while combinators are modeled as functions that take one or more *Search* values, and compute a derived one from that. Although conceptually this is best modeled as a tree structure, with each subtree evaluating to a *Search*, processing happens top-down, and only a single path through the combinator tree is active at a given time. The list of combinators along this path will be called the combinator stack. Figure 2 shows the combinator stack for the earlier branch-and-bound example.

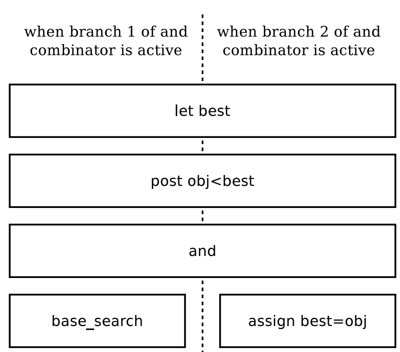

Fig. 2. Branch-and-bound combinator stack

3.3 The Code Generator

Inside *Search* structures, values of type *Gen m* will be built up. They contain a number of hooks that produce the corresponding AST fragments.[5].

As will be explained later, some combinators need to keep an own modifiable state during code generation, so hooks must support side effects; hence *Gen* is parametrized in a monad m.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Gen}\ m = \mathit{Gen}\ \{\mathit{init}_G\ \ :: m\ \mathit{Stmt}, \mathit{body}_G :: m\ \mathit{Stmt}\\
&\qquad\qquad\qquad\quad , \mathit{add}_G\ \ :: m\ \mathit{Stmt}, \mathit{try}_G\ \ :: m\ \mathit{Stmt}\\
&\qquad\qquad\qquad\quad , \mathit{result}_G :: m\ \mathit{Stmt}, \mathit{fail}_G\ \ :: m\ \mathit{Stmt}\\
&\qquad\qquad\qquad\quad , \mathit{height}\ \ :: \mathit{Int}\}
\end{aligned}
$$

The separate hooks correspond to several stages for the processing of nodes in a search tree. Nodes are initialized with init_G and processed using consecutively body_G, add_G, and try_G. result_G is used for reporting solutions, and fail_G for aborting after failure. The *height* field indicates how high the stack of combinators is.

The fragments of the different hooks are combined according to the following template.

$$
\begin{aligned}
&\mathit{gen} :: \mathit{Monad}\ m \Rightarrow \mathit{Gen}\ m \rightarrow m\ \mathit{Stmt}\\
&\mathit{gen}\ g = \textbf{do}\ \mathit{init} \leftarrow \mathit{init}_G\ g\\
&\qquad\qquad\quad \mathit{try}\ \ \leftarrow \mathit{try}_G\ g\\
&\qquad\qquad\quad \mathit{body} \leftarrow \mathit{body}_G\ g\\
&\qquad\qquad\quad \mathit{return}\ \$\ \ \mathit{declarations}\\
&\qquad\qquad\qquad\qquad ; \mathit{init}\\
&\qquad\qquad\qquad\qquad ; \mathit{try}\\
&\qquad\qquad\qquad\qquad ; \mathit{While}\ \mathit{queueNotEmpty}\ \mathit{body}
\end{aligned}
$$

After emitting a number of variable declarations which we omit due to space constraints, the template creates the root node in the search tree through init_G, and try_G initializes a queue with child nodes of the root. Then, in the main part of the algorithm, nodes in the queue are processed one at a time with the body_G hook.

3.4 Code Generation Mixins

Instead of writing a monolithic code generator for every different search heuristic, we modularly compose new heuristics from one or more components, each of which corresponds to a constructor in the high-level DSL. Our code generator components are implemented as (functional) mixins [2], where the result is a function from `Eval m` to `Eval m`, which gets called with its own resulting strategy as argument. The function argument in these mixins is comparable to the *this* object in object-oriented paradigms.

[5] See Section 3.4 for why we partition the code generation into these hooks

type $Mixin\ a = a \rightarrow a$
type $MGen\ m = Mixin\ (Gen\ m)$

There are two kinds of mixin components: *base* components that are self-contained, and *advice* components that extend or modify another component [6]. An alternative analogy for mixins, that includes multi-argument combinators, is that of *inheritance*, where we distinguish self-contained "base classes" and "class deltas". The application of a class delta Δ to a number of classes $\bar{C}$ yields a subclass $\Delta(\bar{C})$; this subclass is said to inherit from $\bar{C}$. When $\bar{C}$ consists of more than one class, we speak of *multiple inheritance*.

Base Component Base searches are implemented as $Gen\ m \rightarrow Gen\ m$ functions (shortened using a type alias to $MGen\ m$ here), with fixpoint semantics. Through lazy evaluation, we can pass the fully combined search as an argument back to itself. Through this mechanism, we can make the base search's hooks call other hooks back at the top of the chain, as shown in the protocol overview shown in Figure 3.

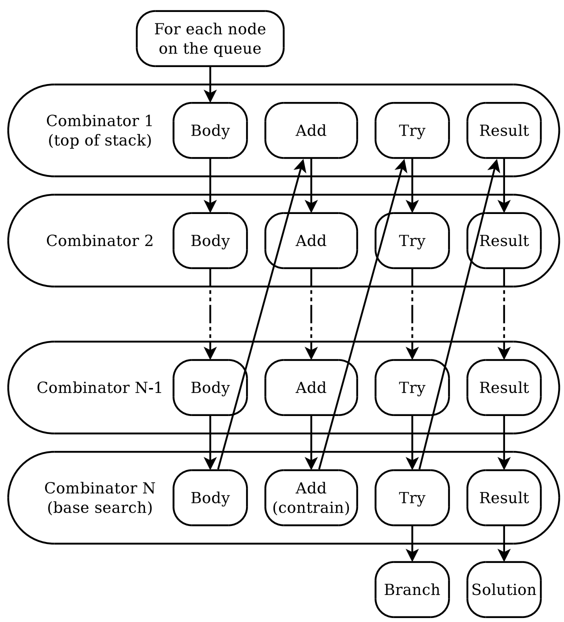

Fig. 3. Node processing protocol

The main example of a base component is the enumeration strategy $base_M$:

$base_M :: Monad\ m \Rightarrow MGen\ m$
$base_M\ this =$

$$
\begin{array}{lll}
\mathit{Gen}\ \{\, \mathit{init}_G & = \mathit{return}\ \mathit{Nop} & \\
\quad ,\, \mathit{body}_G & = \mathit{add}_G\ \mathit{this} & \\
\quad ,\, \mathit{add}_G & = \mathit{constrain} \mathbin{⨾} \mathit{try}_G\ \mathit{this} & \\
\quad ,\, \mathit{try}_G & = \mathbf{let}\ \mathit{ret} & = \mathit{result}_G\ \mathit{this} \\
 & \qquad\quad \mathit{succ} & = \mathit{if}'\ \mathit{isSolved}\ \mathit{ret}\ \mathit{doBranch} \\
 & \qquad \mathbf{in}\ \mathit{if}'\ \mathit{isFailed}\ (\mathit{fail}_G\ \mathit{this})\ \mathit{succ} & \\
\quad ,\, \mathit{result}_G & = \mathit{return}\ \mathit{Nop} & \\
\quad ,\, \mathit{fail}_G & = \mathit{return}\ \mathit{Nop} & \\
\quad ,\, \mathit{height} & = 0\,\} &
\end{array}
$$

The above code omits details related to posting constraints (*constrain*), checking the solver status (*isSolved* or *isFailed*) and branching (*doBranch*). The details of these operations depend on the particular constraint solver involved (e.g. finite domain, linear programming, ...); here we focus only on the search heuristics, which are orthogonal to those details.

As we can see the base component is parametrized by *this*, the overall search heuristic. This way, the base_M search can make the final call to body_G redirect to an add_G on the top of the combinator-stack again, restarting the processing top-down, but this time using add_G instead of body_G. A similar construct is used for called try_G and result_G.

The simplest form of a search heuristic is obtained by applying the fix-point combinator to a base component:

$$
\begin{array}{l}
\mathit{fix} :: \mathit{Mixin}\ a \to a \\
\mathit{fix}\ m = m\ (\mathit{fix}\ m) \\
\mathit{search}_1 :: \mathit{Gen}\ \mathit{Identity} \\
\mathit{search}_1 = \mathit{fix}\ \mathit{base}_M
\end{array}
$$

Advice Component The mixin mechanism allows us to plug in additional advice components before applying the fix-point combinator. This way we can modify the base component's behavior.

Consider a simple example of an advice combinator that prints solutions:

$$
\begin{array}{l}
\mathit{print}_M :: \mathit{Monad}\ m \Rightarrow \mathit{MGen}\ m \\
\mathit{print}_M\ \mathit{super} = \mathit{super}\ \{\, \mathit{result}_G = \mathit{printSolution} \mathbin{⨾} \mathit{result}_G\ \mathit{super} \\
\qquad\qquad\qquad\qquad\quad ,\, \mathit{height} \ \ = 1 + \mathit{height}\ \mathit{super}\,\}
\end{array}
$$

where *printSolution* consists of the necessary solver-specific code to access and print the solution. A code generator is obtained through mixin composition, simply using $(\circ)$:

$$
\begin{array}{l}
\mathit{search}_2 :: \mathit{Gen}\ \mathit{Identity} \\
\mathit{search}_2 = \mathit{fix}\ (\mathit{print}_M \circ \mathit{base}_M)
\end{array}
$$

3.5 Monadic Components

In the components we have seen so far, the monad type parameter m has not been used. It does become essential when we turn to more complex components such as the binary conjunction $\mathsf{and}([g_1, g_2])$.

The code presented at the end of this section shows a simplified *and* combinator, for two *Gen m* structures with the same type m. It does require m to be an instance of *MonadReader Side*, to store the current branch at code-generation runtime. While some hooks simply dispatch to the corresponding hook of the currently active branch, $body_G$ and $result_G$ are more elaborate.

First of all, we also need to store the branch number at program runtime. This is known at the time when the node is created, but needs to be restored into the monadic state when activating it. We assume the functions *store* and *retrieve* give access to a runtime state for each node, indexed with a field name and the height of the combinator involved.

When the $result_G$ hook is called — implying a solution for a sub-branch was found — there are two options. Either the g_1 was active, in which case both the runtime state and the monadic state are updated to In_2, and $init_G$ and try_G for g_2 are executed, which will possibly cause the node to be added to the queue, if branching is required. When this new node is activated itself, its $body_G$ hook will be called, retrieving the branch information from the runtime state, and dispatching dynamically to g_2. When a solution is reached after switching to g_2, $result_G$ will finally call g_2's $result_G$ to report the full solution.

```
data Branch = In1 | In2
type Mixin2 a = a → a → a
and_M :: MonadReader Branch m ⇒ Mixin2 (Gen m)
and_M g1 g2 = Gen { init_G   = store myHeight "pos" In1 ⨟ init_G g1
                  , add_G    = dispatch add_G
                  , try_G    = dispatch try_G
                  , fail_G   = dispatch fail_G
                  , body_G   = myBody
                  , result_G = myResult
                  , height   = myHeight }
  where parent      = ask >>= λx → case x of
                        In1 → return g1
                        In2 → return g2
        dispatch f = parent >>= f
        myHeight   = 1 + max (height g1) (height g2)
        myBody     = let pos = retrieve myHeight "pos"
                         br1 = local (const In1) (body_G g1)
                         br2 = local (const In2) (body_G g2)
                     in if' (pos =:= In1) br1 br2
        myResult   = do num ← ask
                        case num of
                          In1 → local (const In2) $
```

$$
\begin{aligned}
&\qquad\qquad \mathit{store}\ \mathit{myHeight}\ \texttt{"pos"}\ \mathit{In}_2\\
&\qquad\quad ⨾\ \mathit{liftM2}\ (;)\ (\mathit{init}_G\ g_2)\ (\mathit{try}_G\ g_2)\\
&\mathit{In}_2 \to \mathit{result}_G\ g_2
\end{aligned}
$$

3.6 Effect Encapsulation

So far we have parametrized *MGen* with m, a monad type parameter. This parameter will have to be assembled appropriately from monad transformers to satisfy the need of every mixin component in the code generator. Doing this manually can be quite cumbersome. Especially for a large number of mixin components with multiple instances of, e.g., *StateT* this becomes impractical. To simplify the process, we turn to a technique proposed by Schrijvers and Oliveira [11] to encapsulate the monad transformers inside the components.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Search} = \forall t_2.\mathit{MonadTrans}\ t_2 \Rightarrow\\
&\quad \mathit{Search}\ \{\mathit{mgen} :: \forall m\ t_1.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1) \Rightarrow \mathit{MGen}\ ((t_1 \rhd t_2)\ m)\\
&\qquad\qquad ,\mathit{run}\quad :: \forall m\ x.\ \mathit{Monad}\ m \Rightarrow t_2\ m\ x \to m\ x\}
\end{aligned}
$$

To that end we now represent components by the *Search* type that was announced earlier, which packages the components behavior *MGen* with its side effect t_2. The monad transformer t_2 is existentially quantified to remain hidden; we can eliminate it from a monad stack with the *run* field. The hooks of the component are available through the *mgen* field, which specifies them for an arbitrary monad stack in which t_2 is surrounded by more effects t_1 above and m below. Here $t_1 \rhd t_2$ indicates that the focus rests on t_2 (away from t_1) for resolving overloaded monadic primitives such as *get* and *put*, for which multiple implementations may be available in the monad stack. We refer to [12,11] for details of this focusing mechanism, known as the *monad zipper*.

An auxiliary function promotes a non-effectful *MGen m* to *MSearch*:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{MSearch} = \mathit{Mixin}\ \mathit{Search}\\
&\mathit{mkSearch} :: (\forall m.\mathit{Monad}\ m \Rightarrow \mathit{MGen}\ m) \to \mathit{MSearch}\\
&\mathit{mkSearch}\ f\ \mathit{super} =\\
&\quad \mathbf{case}\ \mathit{super}\ \mathbf{of}\\
&\qquad \mathit{Search}\ \{\mathit{mgen} = \mathit{mgen}, \mathit{run} = \mathit{run}\} \to \mathit{Search}\ \{\mathit{mgen} = f \circ \mathit{mgen}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\mathit{run}\quad = \mathit{run}\}
\end{aligned}
$$

which we can apply for instance to base_M and print_M.

$$
\begin{aligned}
&\mathit{base}_S, \mathit{print}_S :: \mathit{MSearch}\\
&\mathit{base}_S\ \ = \mathit{mkSearch}\ \mathit{base}_M\\
&\mathit{print}_S = \mathit{mkSearch}\ \mathit{print}_M
\end{aligned}
$$

Similarly, we define $\mathit{mkSearch}_2$ for lifting binary combinators like and_M. It takes a combinator for two *Gen m*'s, as well as a run function for additional monad transformers the combinator may require, and lifts it to *MSearch2* (implementation omitted).

$$\textbf{type}\ \mathit{MSearch}_2 = \mathit{Mixin}_2\ \mathit{Search}$$

$$\begin{aligned}
&\mathit{and}_S :: \mathit{MSearch}_2\\
&\mathit{and}_S = \mathit{mkSearch}_2\ \mathit{and}_M\ (\mathit{flip}\ \mathit{runReaderT}\ \mathit{In}_1)
\end{aligned}$$

$$\begin{aligned}
\mathit{mkSearch}_2 :: {} & \mathit{MonadTrans}\ t_2\\
& \Rightarrow (\forall m\ t_1.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1) \Rightarrow \mathit{Mixin}_2\ (\mathit{Gen}\ ((t_1 \rhd t_2)\ m)))\\
& \rightarrow (\forall m\ x.\mathit{Monad}\ m \Rightarrow t_2\ m\ x \rightarrow m\ x)\\
& \rightarrow \mathit{MSearch}_2
\end{aligned}$$

Finally we produce C++ code from a *Search* component with *generate*:

$$\begin{aligned}
&\mathit{generate} :: \mathit{Search} \rightarrow \mathit{Stmt}\\
&\mathit{generate}\ s = \textbf{case}\ s\ \textbf{of}\\
&\qquad \mathit{Search}\ \{\mathit{mgen} = \mathit{mgen}, \mathit{run} = \mathit{run}\} \rightarrow\\
&\qquad\quad \mathit{runIdentity}\ \$\ \mathit{run}\ \$\ \mathit{runIdentityT}\ \$\ \mathit{runZ}\ \$\ \mathit{gen}\ \$\ \mathit{fix}\ \$\ \mathit{mgen}
\end{aligned}$$

This code first applies the fix-point computation, passing the result back into itself, as explained earlier. After that, *gen* is called to get the real code-generating monad action. It extracts the knot-tied body_G hook, *runZ* eliminates $\rhd$ from $(t_1 \rhd t_2)\ m$, yielding $t_1\ (t_2\ m)$. Then *runIdentityT* eliminates t_1 (instantiating it to be *IdentityT*), *run* eliminates t_2, and *runIdentity* finally eliminates m (instantiating it to be *Identity*) to yield a *Stmt*.

4 Memoization and Inlining

Experimental evaluation indicates that several component hooks in a complex search heuristic are called frequently, as for example the fail_G hook can be called from many different places. This is a problem 1) for the code generation — which needs to generate the corresponding code over and over again — and 2) for the generated program which contains much redundant code. Both significantly impact the compilation time (in Haskell and in C++); in addition, an overly large binary executable may aversely affect the cache and ultimately the running time.

4.1 Basic Memoization

A well-known approach that avoids the first problem, repeatedly computing the same result, is *memoization*. Fortunately, Brown and Cook [4] have shown that memoization can be added as a monadic mixin component without any major complications.

Memoization is a side effect for which we define a custom monad transformer:

$$\begin{aligned}
&\textbf{newtype}\ \mathbb{M}_T\ m\ a = \mathbb{M}_T\ \{\mathit{run}\mathbb{M}_T :: \mathit{StateT}\ \mathit{Table}\ m\ a\}\\
&\quad \textbf{deriving}\ (\mathit{MonadTrans})
\end{aligned}$$

$$\begin{aligned}
&\mathit{runMemoT} :: \mathit{Monad}\ m \Rightarrow \mathbb{M}_T\ m\ a \rightarrow m\ (a, \mathit{Table})\\
&\mathit{runMemoT}\ m = \mathit{runStateT}\ (\mathit{run}\mathbb{M}_T\ m)\ \mathit{initMemoState}
\end{aligned}$$

which is essentially a state transformer that maintains a table from *Key*s to *Stmt*s. For now we use *String*s as *Key*s.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Key}\ \ = \mathit{String} \\
\textbf{newtype}\ \mathit{Table} = \mathit{Map}\ \mathit{Key}\ \mathit{Stmt} \\
\mathit{initMemoState}\ \ = \mathit{empty}
\end{array}
$$

We capture the two essential operations of $\mathbb{M}_T$ in a type class, which allows us to lift the operations through other monad transformers.[6]

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathbb{M}_M\ m\ \textbf{where} \\
\quad \mathit{get}\mathbb{M} :: \mathit{String} \to m\ (\mathit{Maybe}\ \mathit{Stmt}) \\
\quad \mathit{put}\mathbb{M} :: \mathit{String} \to \mathit{Stmt} \to m\ () \\
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathbb{M}_M\ (\mathbb{M}_T\ m)\ \textbf{where} \ldots \\
\textbf{instance}\ (\mathbb{M}_M\ m, \mathit{MonadTrans}\ t) \Rightarrow \mathbb{M}_M\ (t\ m)\ \textbf{where} \ldots
\end{array}
$$

These operations are used in an auxiliary mixin function:

$$
\begin{array}{l}
\mathit{memo} :: \mathbb{M}_M\ m \Rightarrow \mathit{String} \to \mathit{Mixin}\ (m\ \mathit{Stmt}) \\
\mathit{memo}\ s\ m = \textbf{do}\ \mathit{stm} \leftarrow \mathit{get}\mathbb{M}\ s \\
\qquad\qquad\qquad\quad \textbf{case}\ \mathit{stm}\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nothing} \to \textbf{do}\ \mathit{code} \leftarrow m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{put}\mathbb{M}\ s\ \mathit{code} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ \mathit{code} \\
\qquad\qquad\qquad\qquad \mathit{Just}\ \mathit{code} \to \mathit{return}\ \mathit{code}
\end{array}
$$

which is used by the advice component:

$$
\begin{array}{l}
\mathit{memo}_M :: \mathbb{M}_M\ m \Rightarrow \mathit{MGen}\ m \\
\mathit{memo}_M\ \mathit{super} = \mathit{super}\ \{\ \mathit{init}_G\ \ = \mathit{memo}\ \texttt{"init"}\ \ (\mathit{init}_G\ \ \mathit{super}) \\
\qquad\qquad\qquad\qquad\quad ,\mathit{body}_G = \mathit{memo}\ \texttt{"body"}\ \ (\mathit{body}_G\ \mathit{super}) \\
\qquad\qquad\qquad\qquad\quad ,\mathit{add}_G\ \ = \mathit{memo}\ \texttt{"add"}\ \ (\mathit{add}_G\ \ \mathit{super}) \\
\qquad\qquad\qquad\qquad\quad ,\mathit{try}_G\ \ = \mathit{memo}\ \texttt{"try"}\ \ (\mathit{try}_G\ \ \mathit{super}) \\
\qquad\qquad\qquad\qquad\quad ,\mathit{result}_G = \mathit{memo}\ \texttt{"result"}\ (\mathit{result}_G\ \mathit{super}) \\
\qquad\qquad\qquad\qquad\quad ,\mathit{fail}_G\ \ = \mathit{memo}\ \texttt{"fail"}\ \ (\mathit{fail}_G\ \ \mathit{super})\ \}
\end{array}
$$

which allows us to define, e.g., a memoized variant of print_S.

$$
\mathit{print}_S = \mathit{mkSearch}\ (\mathit{memo}_M \circ \mathit{print}_M)
$$

Note that in order to lift memo_M to a *Search* structure, *Search* must be updated with a $\mathbb{M}_M\ m$ constraint, and *generate* must be updated to incorporate *runMemoT* in its evaluation chain.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Search} = \forall t_2.\mathit{MonadTrans}\ t_2 \Rightarrow \\
\quad \mathit{Search}\ \{\ \mathit{mgen} :: \forall m\ t_1.(\mathbb{M}_M\ m, \mathit{MonadTrans}\ t_1) \Rightarrow \mathit{MGen}\ ((t_1 \rhd t_2)\ m)
\end{array}
$$

[6] For lack of space we omit the straightforward instance implementations.

$$
\begin{array}{l}
\quad\quad , run \quad :: \forall m\ x.\, \mathbb{M}_M\ m \Rightarrow t_2\ m\ x \rightarrow m\ x\}\\
generate\ s =\\
\quad \mathbf{case}\ s\ \mathbf{of}\\
\quad\quad Search\ \{mgen = mgen, run = run\} \rightarrow\\
\quad\quad\quad runIdentity\ \$\ runMemoT\ \$\ run\ \$\ runIdentityT\ \$\ runZ\ \$\ gen\ \$\ fix\ mgen
\end{array}
$$

4.2 Monadic Memoization

Unfortunately, it is not quite this simple. The behavior of combinator hooks may depend on internal updateable state, like and_M from section 3.5 kept a *Branch* value as state. The above memoization does not take this state dependency into account.

In order to solve this issue, we must expose the components' state to the memoizer. This is done in two steps. First, $\mathbb{M}_T$ keeps a *context* in addition to the memoization table, and provides access to it through the $\mathbb{M}_M$ type class. Second — for the specific case of a *ReaderT* s with s an instance of *Showable* — an alternative implementation (*MemoReaderT*) which updates the context in the $\mathbb{M}_T$ layer below it, is provided. Typically, the used states are simple in structure.

To implement this, the *Table* type is extended:

$$
\begin{array}{l}
\mathbf{type}\ MemoContext = Map\ Int\ String\\
\mathbf{type}\ Key \qquad\qquad = (MemoContext, String)\\
\mathbf{data}\ Table = Table\ \{context :: MemoContext\\
\qquad\qquad\qquad\qquad\ , memoMap :: Map\ Key\ Stmt\}\\
initMemoState = Table\ \{context \quad = empty\\
\qquad\qquad\qquad\qquad\quad\ , memoMap = empty\}
\end{array}
$$

MemoContext is represented as a map from integers to strings. The integers are identifiers assigned to the monad transformer layers that have context, and the strings are serialized versions of the contextual data inside those layers (using *show*).

The $\mathbb{M}_M$ type class is extended to support modifying the context information, using *setCtx* and *clearCtx*.

$$
\begin{array}{l}
\mathbf{class}\ Monad\ m \Rightarrow \mathbb{M}_M\ m\ \mathbf{where}\\
\quad ...\\
\quad setCtx :: Int \rightarrow String \rightarrow m\ ()\\
\quad clearCtx :: Int \rightarrow m\ ()
\end{array}
$$

Finally, $\mathbb{MR}_T$ is introduced. It will contain a wrapped double *ReaderT*-transformed monad. The state will be stored in the first, while the second is used to give access to the identifier of the layer.

$$
\mathbf{newtype}\ \mathbb{MR}_T\ s\ m\ a = \mathbb{MR}_T\ \{run\mathbb{MR}_T :: ReaderT\ Int\ (ReaderT\ s\ m)\ a\}
$$

For convenience, $\mathbb{MR}_T$ is made an instance of *MonadReader*, so switching from *ReaderT* to $\mathbb{MR}_T$ does not require any changes to the code interacting with it.

When running a $\mathbb{MR}_T$ transformer, the enclosing *Gen*'s *height* parameter is passed to *rReaderT*, using that as identifier for the layer. The runtime state itself is stored inside the wrapped *ReaderT* layer, while a serialized representation (using *show*) is stored in the context of the underlying $\mathbb{M}_T$. Note that *show* implementations are supposed to turn a value into equivalent Haskell source code for reconstructing the value — this is far from the most efficient solution, but it does produce canonical descriptions for all values, and default implementations are provided by the system for almost all useful data types. There are alternatives, such as using an *Ord*-providing *Dynamic*-like type, but those are harder to implement and there is little to be gained, as will be shown in the evaluation (Section 5).

```
instance (Show s, 𝕄_M m) ⇒ MonadReader s (𝕄ℝ_T s m) where
  ask = 𝕄ℝ_T $ lift ask
  local s m = 𝕄ℝ_T $ do n ← ask
                         old ← lift ask
                         let new = s old
                         putCtx n $ show new
                         let im = run𝕄ℝ_T m
                         r ← mapReaderT (local $ const new) im
                         putCtx n $ show old
                         return r
r𝕄ℝ_T :: (𝕄_M m, Show s) ⇒ s → Int → 𝕄ℝ_T s m a → m a
r𝕄ℝ_T s height m =
  do let action = runReaderT (run𝕄ℝ_T m) height
     putCtx height (show s)
     result ← runReaderT action s
     clearCtx height
     return result
```

4.3 Backend Sharing

So far we have only solved the first performance problem, repeated generation of code. Memoization avoids the repeated execution of hooks by storing and reusing the same C++ code fragment. However, the second performance problem, repeated output of the same C++ code, remains.

We preserve the sharing obtained through memoization in the backend, by depositing the memoized code fragment in a C++ function that is called from multiple sites. Conceptually, this means that a memoized hook returns a function call (rather than a potentially big code fragment), and produces a function definition as a side effect.[7]

$$
\begin{aligned}
&\mathit{memo}_2 :: \mathbb{M}_M\ m \Rightarrow \mathit{String} \rightarrow \mathit{Mixin}\ (m\ \mathit{Stmt})\\
&\mathit{memo}_2\ s\ m = \mathbf{do}\ \mathit{code} \leftarrow \mathit{memo}\ s\ m\\
&\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{name} = \mathit{getFnName}\ \mathit{code}\\
&\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{Call}\ \mathit{name}\ [\,])\\
&\mathit{getFnName} :: \mathit{Stmt} \rightarrow \mathit{String}
\end{aligned}
$$

The following *generate* function produces both the main search code and the auxiliary functions for the memoized hooks. By introducing *runMemoT* in the chain of evaluation functions, the types change, and the result will be of type (*Stmt*, *Table*), since that is returned by *runMemoT*.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{FunDef} = \mathit{FunDef}\ \mathit{String}\ \mathit{Stmt}\\
&\mathit{toFunDef} :: \mathit{Stmt} \rightarrow \mathit{FunDef}\\
&\mathit{toFunDef}\ \mathit{stm} = \mathit{FunDef}\ (\mathit{getFnName}\ \mathit{stm})\ \mathit{stm}\\
&\mathit{generate} :: \mathit{Search} \rightarrow (\mathit{Stmt}, [\mathit{FunDef}])\\
&\mathit{generate}\ s =\\
&\quad \mathbf{case}\ s\ \mathbf{of}\\
&\qquad \mathit{Search}\ \{\mathit{mgen} = \mathit{mgen}, \mathit{run} = \mathit{run}\} \rightarrow\\
&\qquad\quad \mathbf{let}\ \mathit{eval} \qquad\quad\ = \mathit{fix}\ \mathit{mgen}\\
&\qquad\qquad\quad \mathit{codeM} \qquad\ = \mathit{gen}\ \mathit{eval}\\
&\qquad\qquad\quad \mathit{memoM} \qquad = \mathit{run} \circ \mathit{runIdentityT} \circ \mathit{runZ}\ \$\ \mathit{codeM}\\
&\qquad\qquad\quad (\mathit{code}, \mathit{state}) = \mathit{runIdentity}\ \$\ \mathit{runMemoT}\ \mathit{memoM}\\
&\qquad\qquad\quad \mathbf{in}\ (\mathit{code}, \mathit{map}\ \mathit{toFunDef} \circ \mathit{elems}\ \$\ \mathit{memoMap}\ \mathit{state})
\end{aligned}
$$

The result of extracting common pieces of code into separate functions, is shown schematically in figure 4.

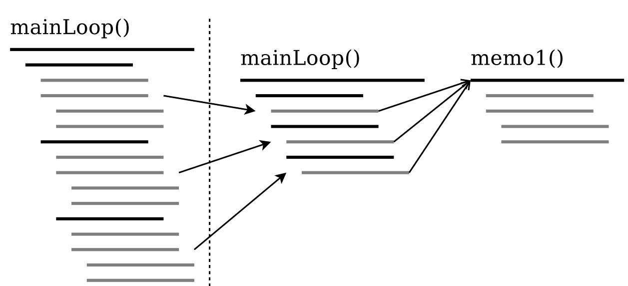

Fig. 4. Memoization with auxiliary functions

[7] The function *getFnName* — given without implementation — derives a unique function name for a given code fragment.

Note that only code generated by the same hook of the same component is shared in a function, not code of distinct hooks or distinct components. Separate from the mechanism described above, it is also possible to detect unrelated *clones* by doing memoization with only the generated code itself as key (instead of function names, present variables and active states). This causes a slowdown, as the code needs to be generated for each instance before it can be recognized as identical to earlier emitted code. To a limited extent, this second memoization scheme is also used in the implementation to reduce the size of generated code — without any measurable overhead.

Finally, applying the above technique systematically results in one generated C++ function per component hook. This is not entirely satisfactory, as many memoized functions are only called once, or only contain a single line of code. One can either rely on the C++ compiler to determine when inlining is lucrative, or perform inlining on the C++ AST in an additional processing step.

5 Evaluation

We have omitted a number of complicating factors in our account, so as not to distract from the main issues. Without going into detail, we list the main differences with the actual implementation:

- There are more hooks, including ones called during branching, adding to the queue, deletion of nodes and switching between nodes belonging to separate strategies. Furthermore, additional hooks exist for the creation of combinator-specific data structures, both globally for the whole combinator, or locally for each node, instead of the dynamic *height*-based mechanism.
- The code generation hooks are functions that take an additional argument, the *path info*. It contains which variable names point to the local and global data structures, which variables need to be passed to generated memoized functions, and pieces of code that need to be executed when the current node needs to be stored, aborted or copied. The values in the path info are also taken into account when memoizing, complicating matters further.
- We have built into the code generators a number of optimizations. For example, if it is known that a combinator never branches, certain generated code and data structures may be omitted.
- Searches keep track of whether they complete exhaustively, or are pruned. Repeat-like combinators use exhaustiveness as an additional stop criterion.

To evaluate the usefulness of our system, benchmarks[8] were performed (see Table 1)[9]. A first set includes the known problems `golfers`[10], `golomb`[11], `open`

[8] Available at `http://users.ugent.be/~tschrijv/SearchCombinators`

[9] A 2.13GHz Intel(R) Core(TM)2 Duo 6400 system, with 2GiB of RAM was used. The system was running Ubuntu 10.10 64-bit, with GCC 4.4.4, Gecode 3.3.1 and Minizinc 1.3.1.

[10] Social golfer problem, CSPlib problem 10

[11] Golomb rulers, CSPlib problem 6

name	size	memo?	lines	hooks	trans.		time		
					eff.	total	generate	build	run
golomb	10	no	216	70	4	14	0.00017	2.0	4.9
		yes	187	95	5	17	0.0073	2.0	4.9
	11	no							110
		yes							110
	12	no							1200
		yes							1200
open-stacks	30	no	216	70	4	14	0.00016	2.1	0.12
		yes	187	95	5	17	0.0074	2.0	0.12
golfers		no	119	29	3	8	0.00017	2.0	1.3
		yes	114	46	4	11	0.00017	2.0	1.3
radiation	15	no	11455	4153	4	76	0.57	16	210
		yes	2193	1155	5	79	0.19	4.0	230
	5	no	2530	898	4	36	0.073	4.3	0.10
		yes	933	485	5	39	0.055	2.7	0.10
bab-real		no	216	70	4	14	0.00019	2.0	17
		yes	187	95	5	17	0.0074	2.0	17
bab-restart		no	1499	1166	5	20	0.045	2.8	17
		yes	433	262	6	23	0.026	2.2	17
for+copy		no	1164	414	5	14	0.016	2.4	8.9
		yes	494	180	6	17	0.0066	2.1	8.9
once-sequence		no	2530	898	4	36	0.073	4.2	2.7
		yes	933	485	5	39	0.054	2.7	2.6
ortest	10	no	1597	849	13	48	0.11	3.2	17
		yes	1222	655	14	51	0.11	2.6	17
	20	no	4232	1869	23	88	0.82	9.7	17
		yes	3352	1465	24	91	0.79	6.7	17

Table 1. Benchmark results

`stacks` and `radiation`[1]; a second set contains artificial stress tests. The different problem sizes for `golomb` use the same search code, while in `ortest` and `radiation`, separate code is used.

The first three columns give the name, problem size and whether or not the memoizing version was used. Further columns show the number of generated C++ lines (col. 4), the number of invoked hooks (col. 5), the number of monad transformers active (both the effective ones (col. 6), and including $IdentityT$ and $\triangleright$ (col. 7)). Finally, the average generation (Haskell, col. 8), build (gcc, col. 9) and run time (col. 10) are listed. All these numbers are averages over many runs (of up to an hour of runtime).

For the larger problem instances, memoization reduces both generation time and build time, by reducing the number of generated lines. No reduced cache effects resulting from memoizing large generated code are observed in these examples, but performance is not affected either by the increased number of function calls. In particular for the `radiation` example, the effect of memoization is drastic. On the other hand, for small problems, memoization does not help, but the overhead is very small.

6 Related Work

We were inspired by the monadic mixin approach to memoization of Brown and Cook [4]. The problem of memoization of stateful monad components is not yet solved in general, but typically requires some way for exposing the implicit state, as shown in [3] for parser combinators. In our system, this is accomplished by also memoizing the implicit state.

A different approach that results in smaller code generated from a DSL is *observable sharing* [5,8]. Yet, the main intent of observable sharing is quite different. Its aim is to preserve sharing at the level of Haskell in the resulting generated code, typically using *unsafePerformIO*. It does not detect distinct calls that result in the same code, and is hard to integrate with code-generating monadic computations as appear in our setting.

Our work is directly inspired by earlier work on the Monadic Constraint Programming DSL [13,15]. In particular, we have studied how to compile high-level problem specifications in Haskell to C++ code for the Gecode library [14]. The present complements this with high-level search specifications.

7 Conclusions

We have shown how to implement a code generator for declarative specification of a search heuristic using monadic mixins. Using this mixin-based approach, search combinators can be implemented in a modular way, and still independently modify the behavior of the generated code. Through existential types and the monad zipper, all combinators can introduce their own monad transformers to keep their own state throughout the code generation, without affecting any other transformers.

Since the naive approach leads to certain hooks being invoked many times over, we turn to memoization to avoid code duplication. Memoization is implemented as another monadic mixin which is added transparently to existing combinators.

The system is implemented as a Haskell program that generates search code in C++ from a search specification in MiniZinc which is then further integrated in a CP solver (Gecode). Our benchmarks demonstrate the impact of memoizing the monadic mixins.

References

1. Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter Stuckey. CP and IP approaches to cancer radiotherapy delivery optimization. *Constraints*, 2011.

2. Gilad Bracha and William R. Cook. Mixin-based inheritance. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 303–311, 1990.
3. Daniel Brown and William R. Cook. Function inheritance: Monadic memoization mixins. Report, Department of Computer Sciences, University of Texas at Austin, June 2006.
4. Daniel Brown and William R. Cook. Function inheritance: Monadic memoization mixins. In *Brazilian Symposium on Programming Languages (SBLP)*, 2009.
5. Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, ASIAN '99, pages 62–73, London, UK, 1999. Springer-Verlag.
6. Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. Effectiveadvice: disciplined advice with explicit effects. In Jean-Marc Jézéquel and Mario Südholt, editors, *AOSD*, pages 109–120. ACM, 2010.
7. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
8. Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM.
9. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessire, editor, *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
10. Horst Samulowitz, Guido Tack, Julien Fischer, Mark Wallace, and Peter Stuckey. Towards a lightweight standard search language. In Justin Pearson and Toni Mancini, editors, *Constraint Modeling and Reformulation (ModRef'10)*, 2010.
11. Tom Schrijvers and Bruno Oliveira. Modular components with monadic effects. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*, number UU-CS-2010-020, pages 264–277, 2010.
12. Tom Schrijvers and Bruno Oliveira. The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven, 2010.
13. Tom Schrijvers, Peter J. Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, 2009.
14. Pieter Wuille and Tom Schrijvers. Monadic Constraint Programming with Gecode. In *Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation*, pages 171–185, 2009.
15. Pieter Wuille and Tom Schrijvers. Parametrized models for on-line and off-line use. In J. Marino, editor, *WFLP 2010 Post-Proceedings*, LNCS. Springer, 2011.