Structure-Preserving Compilation: Efficient Integration of Functional DSLs into Legacy Systems

Julian Kranz
Technische Universität München, Institut für Informatik II, Garching, Germany
julian.kranz@in.tum.de

Axel Simon
Technische Universität München, Institut für Informatik II, Garching, Germany
axel.simon@in.tum.de

Abstract

We present a translation scheme from a pure functional domain-specific language (DSL) to C. The over-arching idea of this scheme is to preserve the structure of the input program as much as possible. This includes, among other things, to refrain from inlining user-written functions and to retain variable names as much as possible. We apply this translation scheme to GDSL, a DSL used for the specification of decoders for machine instructions. GDSL offers non-trivial language features such as monadic actions that our translation scheme maps one-to-one to C statements, resulting in code that closely resembles hand-written C code. Indeed, it is feasible to debug and profile the program at the C level and to interface the generated code with existing C code without marshaling data. Our translation scheme is therefore an attractive starting point for a light-weight DSL since no other language-specific tools besides the compiler are necessary. Moreover, the generated code is amenable to compiler optimizations found in off-the-shelf C compilers. This is illustrated by the performance achieved by a decoder for x86 machine instructions implemented in GDSL which is as fast as a production-quality decoding library shipped by Intel.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages; D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Memory management (garbage collection), Optimization; F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.3.3 [*Studies of Program Constructs*]: Type structure; D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Algorithms, Measurement, Performance, Languages, Theory

Keywords Domain specific languages, translation scheme, interfacing between programming languages, code optimization, functional languages, type inference, instruction decoding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPDP '14, September 8–10, 2014, Canterbury, UK.
Copyright © 2014 ACM 978-1-4503-2947-7/14/09. . . $15.00.
http://dx.doi.org/10.1145/2643135.2643157

1. Introduction

Advances in programming languages are difficult to transfer into an existing product as rewriting existing software using a modern language is usually not cost effective. Furthermore, since interfacing with modern languages is usually problematic, programmers fall back to using legacy programming languages for development: besides the technical challenges of marshaling data and transferring ownership of memory, there are practical maintenance problems in building, debugging, porting and profiling multi-language projects. An interesting alternative for reaping some benefits of higher-level languages is to develop a domain-specific language (DSL) that is translated into the legacy programming language [7]. One example are Cobol programs that nowadays feature embedded SQL statements that a pre-processor translates into the corresponding primitives. A general framework for creating such extensions is provided by the Xtext framework [1] that targets the Java programming language, thereby providing access to the garbage-collected heap and the introspection facilities of Java. In this work, we address the challenge of bridging a wider semantic gap, namely from a pure functional language to the C language so that we can neither rely on a garbage-collected heap nor on run-time type information.

Language features of our DSL such as boxed values, polymorphic functions, flexible records (where fields can be added to an existing record), monads, and closures thus have to be implemented explicitly using C data structures which generally clobber the logic of the generated code with boilerplate implementation details. The observation in this work is that these features are often not used to their full extent in which case they can be compiled into C code that resembles hand-written code. Based on this observation, we combined the following three optimizations into one highly efficient translation scheme.

1.1 Heap-Allocation and Avoidance of Garbage Collection

We solve the lack of a garbage-collected heap by equipping each translated DSL program with a micro-runtime that provides a heap. During the evaluation of a DSL function, objects may be allocated on this heap. The returned result may also contain pointers to the DSL heap. Once the C program has processed the result (possibly but not necessarily by copying the result to `malloc`'d memory regions), a function `reset_heap()` can be called that discards all allocated data on the DSL heap. This operation can be as simple as setting the *free*-pointer to the beginning of the DSL heap

and is thus a very cheap way of reclaiming the memory. Moreover, since no actual garbage collector is needed that traces the reachable set, the objects on the heap do not have to be tagged to distinguish pointers from data. Hence, an object stored on the heap has the same C type as when stored in a stack-allocated variable. As a consequence, code accessing the heap resembles hand-written C code. Moreover, the initialization of newly allocated objects is cheaper and the memory consumption is lower compared to garbage-collected languages that use tagged heap objects.

A consequence is that a long-running DSL program has to be broken down into a number of smaller, isolated tasks between which the heap is reclaimed so that the overall memory usage remains reasonable. These smaller tasks need to be designed so that they do not communicate heap-allocated data among one another. While this may seem as a major restriction, it fits many uses of DSLs [5, 8].

1.2 Unboxing of Polymorphic Values and Closures

Our DSL allows for parametric polymorphism [3] where functions and data types can be defined that are agnostic to the data they operate on, such as a list reversal function. Translating such a function into a single C function requires that polymorphic values are represented uniformly, which usually means that they are represented by a pointer to the heap where the actual, variable-sized value resides. Values stored on the heap are called *boxed*. The efficiency of a program can be greatly improved by *unboxing* values that have a fixed type [12]. For instance, computing `x+(y+z)` without unboxing would allocate the intermediate result i of `y+z` on the heap before it is read from the heap to compute `x+`i. An unboxing optimization avoids this unnecessary allocation step.

The type information necessary to perform unboxing is usually obtained in the front-end through type checking/inference and must be retained up to the unboxing optimization pass which complicates the compiler [15]. Also, a polymorphic function is often only used in a restricted context. For instance, consider sorting a list of tuples `xs : (int*char) list` by the first component in ML:

```
let fun cmp (xKey,xVal) (yKey,yVal) = xKey <= yKey
in sort cmp xs
```

Here, `cmp` is polymorphic in the second tuple element which therefore cannot be unboxed even if it is only used for the type `char`. Indeed, it has been proposed to specialize polymorphic functions for the sake of more unboxing [2]. Our type-refinement optimization unboxes all variables that are used monomorphically such as `cmp` above. For example, if all the lists in a program range over the same type, the payload of the list constructor `Cons` is unboxed. Moreover, it does not require any type information from the front-end and thus also works for dynamically typed DSLs.

Since the monomorphic types used in our optimization can be different from the types inferred in the front-end, we are able to infer additional information on function types and record types. Specifically, a function type carries a tag that indicates if it represents only trivial closures, that is, closures that only contain a function pointer but no arguments. We unbox these closures by passing the address of the function instead of passing the pointer to the heap-allocated closure. With respect to record types, note that our language features flexible records where fields can be added to and extracted from an existing record. These records are realized in the runtime by heap-allocating a linked list of field/value pairs. For readability and for the sake of calling functions with record arguments from C, it is desirable to use C `struct`s instead. To this end, the optimization infers a set of fields (and their types) and whether these fields always exist. Any variable of record type whose fields always exist is turned into a C `struct`.

1.3 Transformation of Monadic Functions

A monadic function f is of type $\alpha_1 \times \ldots \times \alpha_n \to M\ \alpha$ where $M\ \alpha$ is itself an abbreviation for $s \to s \times \alpha$, that is, a function from state s to a new state s and the auxiliary result α. Thus, the evaluation of `f a1 ... an` yields yet another function that expects a state. Since creating, returning and invoking a closure for each monadic function is hard to read and expensive, we transform the type of f to $s \times \alpha_1 \times \ldots \times \alpha_n \to s \times \alpha$ if all of its call sites are monadic, that is, they all supply $\alpha_1 \ldots \alpha_n$ as well as s. Due to this transformation, most monadic functions are translated into normal C functions and many closures are avoided.

In summary, the presented translation scheme contributes the following novelties:

- It illustrates how structure preserving translation can turn a high-level DSL language into a readable low-level legacy language such as C, thereby making it possible to re-use existing debuggers, profilers and other tools for C.
- A front-end independent unboxing optimization is proposed and evaluated that is able to unbox closures and flexible records.
- Our evaluation shows that the generated code can be optimized by off-the-shelf C compilers, yielding code that is as fast as a hand-written C library.

The paper is organized as follows: Section 2 presents a functional DSL called GDSL for assembler instruction decoder specification [11] which serves as a reference language throughout the paper. Section 3 illustrates how our DSL language translates to an imperative language Imp. Section 4 discusses the optimization of Imp. Our implementation, described in Sec. 5, is evaluated in Sec. 6. Related work is presented in Sec. 7 before Sec. 8 concludes.

2. A Domain Specific Language for Instruction Decoding

The purpose of the presented DSL is to ease the specification of instruction decoders which are programs that convert a byte stream to an abstract syntax tree (AST) that represents the assembler instruction [14]. The DSL is also used to compile these instructions to an optimized intermediate representation [11]. Like ML and Haskell, the language features pattern matching on algebraic data structures which eases the symbolic computation necessary when inspecting ASTs. The language is Turing-complete by allowing arbitrary recursion. The domain-specific part of the language are *decoder declarations* featuring a pattern-match syntax that implicitly reads from the input byte stream and that allows calls to other decoders [14]. We illustrate them using an example from the Intel x86 architecture. We allow a slash `/` to be part of an identifier in order to accommodate the Intel nomenclature that uses `/r` in order to describe a register argument.

The minimal decoder in Fig. 1 declares an algebraic data type to represent instructions before defining two decoders `decode` and `/r` which use the special pattern syntax in `[...]`

```
1  # declare a datatype for instructions
2  type insn = ADD of {opnd1:opnd,opnd2:opnd}
3            | INC of {opnd1:opnd} #increment
4
5  # define decoding rules for streams starting
6  # with 0x00 and 0x01; the latter queries the
7  # $opndsz flag of the monadic state
8  val decode [0x00 /r] = binop ADD r/m8 r8
9  val decode [0x01 /r] = do
10   opndsz <- query $opndsz;
11   if opndsz then binop ADD r/m16 r16
12             else binop ADD r/m32 r32
13 end
14
15 # a sub-decoder reading one byte and storing 2
16 # (mod), 3 (reg/opcode) and 3 (rm) bits of it
17 # in the monadic state
18 val /r ['mod:2 reg/opcode:3 rm:3'] =
19   update @{mod=mod, reg/opcode=reg/opcode,
20            rm=rm}
21
22 # a function that generates two-argument
23 # instructions; giveOpX are monadic functions,
24 # cons is an instruction constructor
25 val binop cons giveOp1 giveOp2 = do
26   op1 <- giveOp1;
27   op2 <- giveOp2;
28   return (cons { opnd1=op1, opnd2=op2 })
29 end
30
31 # a typical function passed as argument to
32 # binop: it queries the value of the reg/opcode
33 # field in the monadic state and returns an AST
34 # in the form of an algebraic data type, here a
35 # register REG XX
36 val r16 = do
37   r <- query $reg/opcode;
38   case r of
39      '000': return (REG AX)
40    | '001': return (REG CX)
41    | '010': return (REG DX)
42    | '011': return (REG BX)
43   end # other cases omitted
```

Figure 1. A minimal decoder for Intel x86 instructions.

to indicate that they read from the internal byte stream. This pattern syntax is desugared to the code in Figure 2. Here, the `consume8` is used to extract a byte from the input stream over which a `case`-statement dispatches. The pattern `0x00` is translated into bit-pattern `'00000000'` which is a built-in data type containing the bit-string and its size. The use of the sub-decoder `/r` in both patterns is translated into a call to `/r` in lines 5 and 9, before the right-hand-side of the decode rules are evaluated. Inside `/r`, another byte is read and its bits are extracted using a built-in function `slice`.

The desugared code of `decode` and `/r` in Fig. 2 and the remaining code in Fig. 1 are written in a pure functional language that we call Core. It is defined by the grammar in Fig. 5. A special feature of Core are flexible records which are sets of field/value pairs. Suppose that `r` denotes the record `{f=42, g="g"}`. An update `@{g="h", h='01'}` is a function that replaces or adds fields to a record, i.e. `@{g="h", h='01'} r` evaluates to `{f=42, g="h", h='01'}`.

```
1  val decode = do
2    tok <- consume8;
   # read byte from stream
3    case tok of
   # make pattern matching explicit
4      '00000000' : do
5        /r;
6        binop ADD r/m8 r8
7      end
8    | '00000001' : do
9        /r;
10       opndsz <- query $opndsz;
11       if opndsz
12         then binop ADD r/m16 r16
13         else binop ADD r/m32 r32
14     end
15   end
16 end
17 val /r = do
18   tok <- consume8;
19   rm <- slice tok 0 3;

20   reg/opcode <- slice tok 3 3;
21   mod <- slice tok 6 2;
22   update @{mod = mod,
23            reg/opcode = reg/opcode,
24            rm = rm}
25 end
```

Figure 2. Desugaring the decoders to Core. We omit code handling pattern match failures.

A field `f` is extracted using `$f r` where `$f` is called a field selector. Finally, the language features a built-in state monad. It borrows the `do ...` syntax from Haskell to specify sequences of monadic actions. In contrast to Haskell, the monad is fixed with three pre-defined functions: `query f` applies `f` to the internal state and yields the result, `update f` applies `f` to the internal state, and `return v` merely yields `v`. Note that `return` does not alter the control flow but only serves to return the value `v` as the result of a monad.

The translation scheme and the optimizations developed in this paper allow a translation of Core into the C code shown in Fig. 3 that is sufficiently similar to allow the user of the DSL to debug or profile the program at the C level. Note that this close correspondence is only possible if the program does not exploit the whole flexibility of the source language (which is usually the case). For instance, none of the arguments to `binop` is a partially applied function. Due to this, the address of a C function can be passed as argument as done in line 15, 20, and 22, rather than a pointer to a heap-allocated closure. Also, the arguments of the `REG` constructor in lines 45-52 are preprocessor-defined constants instead of pointers to heap allocated constructors which would be required if one of the registers had an argument. In order to apply these ideas we translate Core to an imperative language Imp.

3. Conversion to an Imperative Language

This section details the conversion from the functional language Core to a language we call Imp whose grammar is given in Fig. 6. Imp is imperative in that the body of a function is a **Block** that contains a sequence of state-

```c
typedef struct /* struct for parameter of ADD */
  {obj_t opnd1; obj_t opnd2;} struct1_t;
typedef struct /* a variant of datatype insn */
  {int_t tag; struct1_t payload;} con_struct1_t;
static inline /* constructor function ADD */
  obj_t constructor_ADD(struct1_t arg_of_ADD) {
  return alloc_con_struct1(
      (con_struct1_t) {1/* ADD */, arg_of_ADD});
}
obj_t decode() {                    /* decode */
  int_t tok = consume8();
  switch (slice(tok, 0, 8)) {
    case 0: /* '00000000' */ {
     _slash_r(); /* call sub-decoder /r */
     return binop(constructor_ADD,r_slash_m8,r8);
    }; break;
    case 1: /* '00000001' */ {
     _slash_r(); /* call sub-decoder /r */
     /* query mondic state */
     int_t opndsz = state.opndsz;
     if (opndsz) return binop(constructor_ADD,
                              r_slash_m16,r16);
       else return binop(constructor_ADD,
                         r_slash_m32,r32);
    }; break;
  };
}
static void _slash_r() {                /* /r */
  int_t tok, rm, reg_slash_opcode, mod;
  tok = consume8(); /* get the next token */
  rm = slice(tok, 0, 3); /* cut out lower 3 bits */
  reg_slash_opcode = slice(tok, 3, 3);
  mod = slice(tok, 6, 2);
  state.mod = mod;
  state.reg_slash_opcode = reg_slash_opcode;
  state.rm = rm;
}
static obj_t constructor_REG(int_t arg_of_REG) {
  return alloc_con_int(
      (con_int_t) {3/* REG */, arg_of_REG});
}
static obj_t r16() {                   /* r16 */
  int_t r = state.reg_slash_opcode;
  switch (slice(r, 0, 3)) {
    case 0: /* '000' */ {
       return constructor_REG(CON_AX); }; break;
    case 1: /* '001' */ {
       return constructor_REG(CON_CX); }; break;
    case 2: /* '010' */ {
       return constructor_REG(CON_DX); }; break;
    case 3: /* '011' */ {
       return constructor_REG(CON_BX); }; break;
  };
}
/* binop expects a pointer to the
   constructor function as parameter */
static obj_t binop(obj_t (*cons)(struct1_t),
    obj_t (*giveOp1)(), obj_t (*giveOp2)()) {
  obj_t op1 = giveOp1();
  obj_t op2 = giveOp2();
  return /* build parameter, call constructor */
    cons((struct1_t){.opnd1=op1,.opnd2=op2}); }
```

Figure 3. The C code of the decoders. Some lines and variable declarations are rearranged for presentational purposes.

Core	::=	*val id id** *=* **Expr**
Action	::=	*id* ← **Expr** *;* **Action**
	|	**Expr** *;* **Action**
	|	**Expr**
Cases	::=	**Pat** *:* **Expr** *;* **Cases** | ε
Pat	::=	*'* (*.* | *0* | *1*)* *'* | *num*
	|	*con id* | *con*
Expr	::=	*let* **Core** *in* **Expr** *end*
	|	*if* **Expr** *then* **Expr** *else* **Expr**
	|	*case* **Expr** *of* **Cases** *end*
	|	**Expr Expr**$^+$
	|	*{* (*field* = **Expr**)* *}*
	|	*@ {* (*field* = **Expr**)* *}*
	|	*$ field*
	|	*query* **Expr** | *update* **Expr**
	|	*do* **Action** *end*
	|	*"* *string* *"* | *'* (*0* | *1*)* *'* | *num*
	|	*con* | *id*

Figure 4. The input language Core.

Imp	::=	*id* (**Decl***) *:* **Type** = *id where* **Block**
	|	*constructor id* (**Decl**)
	|	*closure cid* [**Decl***] (**Decl***) *:* **Type** → *id*
Decl	::=	**Type** *id*
Stmt	::=	(*id* =)? **Expr** *;*
	|	*if* **Expr** *then* **Block** *else* **Block** *;*
	|	*case* **Expr** *of* **Cases** *;*
Block	::=	(**Decl** *;*)* **Stmt***
Cases	::=	*num*$^+$ *:* **Block** *;* **Cases** | ε
Expr	::=	*id* | *num* | "*string*"
	|	**Expr** (**Expr**$^+$)
	|	*** **Expr** (**Expr**$^+$)
	|	*{* (*field* = **Expr**)* *}*
	|	*@ {* (*field* = **Expr**)* *}* **Expr**
	|	*$ field* **Expr**
	|	*box* (**Type** , **Expr**)
	|	*unbox* (**Type** , **Expr**)
	|	*gen-closure cid* [**Expr***]
	|	*λs →* **Block Expr**
	|	*exec* **Expr**

Figure 5. The intermediate language Imp. Note that the non-terminal **Type** is defined in Fig. 8.

ments **Stmt** . Imp is meant to be lowered directly to C. As such, record operations are no longer first class citizens but expressions that require a record as argument. Moreover, Core makes all allocations on the heap explicit. For values, $box(t, e)$ returns a pointer to a freshly allocated heap region containing e while $unbox(t, p)$ returns the value pointed to by p. In both cases, the type $t \in \mathbf{int}, \mathbf{obj}, \ldots$, determines the size of the object. A function f is boxed by creating a closure on the heap using *gen-closure* $f_{cl}[a_1 \ldots a_k]$ and unboxed by invoking the closure using $*p(a_{k+1} \ldots a_n)$ (note the star: a normal function call is written $p(\ldots)$). Here, a_i with $i \leq k$ is a value from the environment in which f was defined. We illustrate this using the following Core program:

```
val f x =
  let
```

```
3     val g y = x+y
4   in
5     g
6   end
7 val six = (f 1) 5
```

The corresponding Imp code looks as follows:

```
1 f(obj x) : obj = fRes where
2   fRes = gen-closure g_cl[x]
3 g(obj x, obj y) : obj = gRes where
4   gRes = box(int,+(unbox(int,x),unbox(int,y)))
5 closure g_cl[obj](obj y): obj -> g
6 six() : obj = sixRes where
7    sixRes = * (f(box(int,1))) (box(int,5))
```

The Core program defines a function f that locally defines g. Since g refers to the variable x from the environment, the Imp program defines g to take x in addition to its parameter y. The value g returned by f in the Core program is translated by computing a closure that contains a pointer to g_{cl} as well as x where g_{cl} is a special declaration that is later translated to code invoking g with the arguments in the passed-in closure. Specifically, the call `f(box(int,1))` in line 7 of the Imp program, returns an address of a closure on the heap, say a_{cl}, that contains a pointer to g_{cl} and a pointer to an integer 1. The invocation `* `a_{cl} `(box(int,5))` calls g_{cl} and passes a_{cl} and the argument `box(int,5)` to it. The C code emitted for g_{cl} in line 5 then invokes g with the argument 1 from the closure and the actual argument 5.

The translation scheme for all Core constructs (except `case`) is formalized in Fig. 4. There is a schema $[\![\cdot]\!]$ for translating a function, for blocks $[\![\cdot]\!]^B_\rho$, for expressions $[\![\cdot]\!]^E$ and for monadic sequences $[\![\cdot]\!]^S$, some of which have side-effects annotated in bold. For instance, the translation of a function f creates a closure declaration f_{cl} and requires that all occurrences of f in the generated program are replaced by the expression that generates a closure of f. (A special case applies for function definitions with no arguments where each call-site f is replaced with $f\,()$. This special case is omitted since it is an artifact of Core where constants and functions are not distinguished.) The schema for translating e into a basic block $[\![e]\!]^B_v$ takes an additional variable v to which the result of evaluating e is assigned to. The translation of an expression $[\![e]\!]^E$, in turn, returns a tuple $\langle \vec{s}, e' \rangle$ where $\vec{s}$ is a sequence of statements that need to be executed in order to compute e whose result is given by the Imp-expression e'. As an example, consider applying the translation of the conditional in Fig 4 on lines 11–13 of Fig. 2. The translation first computes the code of the two branches, given by $[\![e_t]\!]^B_{ite}$ and $[\![e_e]\!]^B_{ite}$, so that the result is stored in the freshly created variable *ite*. Based on these two blocks, the tuple $\langle s, ite \rangle$ is returned where s is the single Imp-statement `if opndz then ite = ...; else ite = ...;`.

Recall that the record selector $\$foo$ in Imp requires an argument, e.g. $\$foo\ r$, so that it can be emitted as the C code `r.foo`. The translation therefore replaced $\$foo$ with a closure of *select-foo* which is a newly generated function that extracts this field from a record. Creating named functions instead of the traditional approach of using an anonymous function $\lambda r \rightarrow \$foo\ r$ [10] has the advantage that a duplicate generation of *select-foo* is easily avoided by checking whether a function with that name already exists and that the resulting C code is easier to read. One speciality is the translation of algebraic data types: a constructor that takes no arguments is translated as a pointer to an integer that stores a tag identifying it. A constructor that does take an argument is translated into a special top-level *constructor* function that heap-allocates an integer for its tag followed by the payload.

3.1 Translating Monadic Sequences

The Core language provides a state monad, that is, a monad that threads a state through a sequence of monadic actions. An action may query or update this state using the so-named primitives. The `do`-notation is borrowed from Haskell and is syntactic sugar for combining the actions in a sequence:

```
1 val f x y = do
2   a <- actA
3   b <- actB
4   actC
```

```
1 val f x y =
2   actA >>= (λa →
3     actB >>= (λb →
4       actC))
```

Here, the type of `actA : M r` is a monad where `a : r` is the type of the result. In our language, `M r` is syntactic sugar for $s \rightarrow \langle r, t \rangle$, that is, our monad has an internal state (of type s) that each action may modify to type t. Two actions are combined using the infix "bind" operator `>>=` as follows:

```
val (act >>= cont) = λs →
    let val <a,t> = act s in cont a t end
```

Compiling monadic code into efficient code that does not generate any closures requires that `>>=` and the λ-term in its right argument are suitably inlined. Not only is the desugared `do`-notation hard to read, avoiding generating closures for the λ-expressions also requires inlining of `>>=` and β-reduction (evaluation of λ-terms). These optimizations destroy the structure of the input program and may lead to excessive code duplication if not applied carefully. Thus, we propose a less literal translation by observing that the state `s` in the definition of `>>=` is not used once `act` is evaluated and the new state `t` is produced. Hence, we use a global variable S to store the state and use monadic functions that, instead of returning a tuple like `act`, destructively update S and only return the result `a`. This translation preserves the pure functional semantics of the input program because the actions of a `do`-sequence are emitted in the order in which they appear in the source program.

The translation scheme in Fig. 4 makes this idea explicit. The `do` statement is translated literally to a special lambda abstraction $\lambda S \rightarrow b\ e$ where b is the **Block** representing the body of the `do` and e is the last expression in the sequence. The $[\![\cdot]\!]^S$ scheme translates each action e by wrapping it with *exec* e which applies e to the global state S. Indeed, *exec* e could be written e S but we chose the former notation for the sake of readability. The result of e S is the result of the monadic action, that is, no new state is returned. Thus, all built-in monadic functions have to use destructive assignments to S to update the state. The only function that modifies the state is *update* e, which is translated into the statement $S = *e'(S)$; and otherwise behaves as an action $\lambda S \rightarrow box(\mathbf{int}, 0)$ that returns the dummy value zero. The *query* e action merely invokes e on S and returns the result. We apply the translation scheme to the lines 8–13 of `decode` in Fig. 2, yielding the following code (note that `r/m16`, etc. are constants that are turned into calls with no arguments):

```
1 decode(): obj = decodeRes where
2   decodeRes = λS → # case statement omitted
3     obj opndsz; obj ite;
4     opndsz = exec (λS →
5             *(gen-closure select_opndsz_cl)(S));
6     if unbox(int,opndsz) then
```

function translation: explicitly add environment variables c_i to parameter list, translate body, declare closure, redirect calls to f to a newly generated closure of f

$[\![val\ f\ x_1 \dots x_n = e]\!] = f(\mathbf{obj}\ c_1 \dots \mathbf{obj}\ c_k\ \mathbf{obj}\ x_1 \dots \mathbf{obj}\ x_n) : \mathbf{obj} = f_r\ where\ [\![e]\!]^B_{f_r}$
declare *closure* $f_{cl}[\mathbf{obj}\ c_1 \dots \mathbf{obj}\ c_k](\mathbf{obj}\ x_1 \dots \mathbf{obj}\ x_n) : \mathbf{obj} \to f$ where $c_i \in free(e)$
apply substitution $f/gen\text{-}closure\ f_{cl}[c_1 \dots c_k]$ **in all bodies**

translate expression e into block, assigning the result of e to x

$[\![e]\!]^B_x = \mathbf{obj}\, v_1 \dots \mathbf{obj}\, v_n; \vec{s}\ x = e';$ where $\langle \vec{s}, e' \rangle = [\![e]\!]^E$ and $\{v_1, \dots v_n\} = free(e)$

let translation: evaluate b, assign result to x, evaluate the body e

$[\![let\ x = b\ in\ e\ end]\!]^E = \langle \vec{s}_b\ x = b'; \vec{s}_e, e' \rangle$ where $\langle \vec{s}_b, b' \rangle = [\![b]\!]^E$ and $\langle \vec{s}_e, e' \rangle = [\![e]\!]^E$

if translation: evaluate condition e_c by executing $\vec{s}_c$, fetch result from heap, compute value of then or else branch into the fresh variable *ite*, return *ite* as result

$[\![if\ e_c\ then\ e_t\ else\ e_e]\!]^E = \langle \vec{s}_c\ if\ unbox(\mathbf{int}, e'_c)\ then\ [\![e_t]\!]^B_{ite}\ else\ [\![e_e]\!]^B_{ite}; , ite \rangle \wedge \langle \vec{s}_c, e'_c \rangle = [\![e_c]\!]^E$ *ite* fresh

closure invocation: evaluate function expression and arguments; invoke function closure

$[\![e_0\ e_1 \dots e_n]\!]^E = \langle \vec{s}_0 \dots \vec{s}_n, *e'_0(e'_1 \dots e'_n) \rangle$ where $\langle \vec{s}_i, e'_i \rangle = [\![e_i]\!]^E$ for $i = 0, n$

record constant: evaluate field expressions e_i before returning the record value $\{f_0 = e'_0 \dots f_n = e'_n\}$, set fields f_i to resulting values

$[\![\{f_0 = e_0 \dots f_n = e_n\}]\!]^E = \langle \vec{s}_1 \dots \vec{s}_n, \{f_0 = e'_0 \dots f_n = e'_n\} \rangle$ where $\langle \vec{s}_i, e'_i \rangle = [\![e_i]\!]^E$

record update: evaluate new field values e'_i, generate an update function $update\text{-}f_1 \cdots f_n$ and its closure upd_{cl}; return a new closure containing the field values

$[\![@\{f_0 = e_0 \dots f_n = e_n\}]\!]^E = \langle \vec{s}_1 \dots \vec{s}_n, gen\text{-}closure\ upd[e'_0 \dots e'_n] \rangle$ where $\langle \vec{s}_i, e'_i \rangle = [\![e_i]\!]^E$
declare *closure* $upd_{cl}[\mathbf{obj}\ c_1 \dots \mathbf{obj}\ c_n](\mathbf{obj}\ r) : \mathbf{obj} \to update\text{-}f_1 \cdots f_n$
declare $update\text{-}f_1 \cdots f_n(c_1 \dots c_n\ r) : \mathbf{obj} = r'\ where\ r' = @\{f_1 = c_1 \dots f_n = c_n\}\ r$

record selector function: generate a function *select-f* and its closure; return a new closure as result

$[\![\$f]\!]^E = \langle \emptyset, gen\text{-}closure\ sel_{cl}[] \rangle$
declare *closure* $sel_{cl}[](\mathbf{obj}\ r) : \mathbf{obj} \to select\text{-}f$
declare $select\text{-}f(\mathbf{obj}\ r) : \mathbf{obj} = r'\ where\ r' = \$f\ r$

query translation: invoke function e on the monadic state and return the result

$[\![query\ e]\!]^E = \langle \vec{s}, \lambda \mathrm{S} \to *e'(\mathrm{S}) \rangle$ where $\langle \vec{s}, e' \rangle = [\![e]\!]^E$

update translation: destructively set the monadic state to $*e'(\mathrm{S})$ and return a dummy value

$[\![update\ e]\!]^E = \langle \vec{s}\ \mathrm{S} = *e'(\mathrm{S}); , \lambda \mathrm{S} \to box(\mathbf{int}, 0) \rangle$ where $\langle \vec{s}, e' \rangle = [\![e]\!]^E$

do translation: return the empty statement sequence and the expression $\lambda \mathrm{S} \to \vec{d};\ \vec{s}\ e$ that evaluates to a closure containing declarations $\vec{d}$, statements $\vec{s}$, and the resulting expression e of the monadic sequence

$[\![do\ seq]\!]^E = \langle \emptyset, \lambda \mathrm{S} \to \vec{d};\ \vec{s}\ e \rangle$ where $\langle \vec{d}, \vec{s}, e \rangle = [\![seq]\!]^S$

translation of assignment within monadic sequence: add declaration of x to assignments $\vec{d}$, evaluate e and apply it to the global state, assign the result to x; append translation of remaining sequence

$[\![x \leftarrow e;\ seq]\!]^S = \langle \vec{d}\ \mathbf{obj}\ x, \vec{s_e}\ x = exec\ e';\ \vec{s}, \bar{e} \rangle$ where $\langle \vec{s_e}, e' \rangle = [\![e]\!]^E \wedge \langle \vec{d}, \vec{s}, \bar{e} \rangle = [\![seq]\!]^S$

translation of expression within monadic sequence: evaluate e, apply it to the global state, append translation of remaining sequence

$[\![e;\ seq]\!]^S = \langle \vec{d}, \vec{s_e}\ exec\ e';\ \vec{s}, \bar{e} \rangle$ where $\langle \vec{s_e}, e' \rangle = [\![e]\!]^E \wedge \langle \vec{d}, \vec{s}, \bar{e} \rangle = [\![seq]\!]^S$

translation of expression at the end of a monadic sequence: evaluate e, apply it to the global state; use the result of the application as result for the sequence

$[\![e]\!]^S = \langle \emptyset, \vec{s}, exec\ e' \rangle$ where $\langle \vec{s}, e' \rangle = [\![e]\!]^E$

translation of string-, integer- and bit vector-constants: return the respective boxed value

$[\!["string"]\!]^E = \langle \emptyset, "string" \rangle$ $\quad [\![n]\!]^E = \langle \emptyset, box(\mathbf{int}, n) \rangle$ $\quad [\!['0101']\!]^E = \langle \emptyset, box(\mathbf{bits}[4], 5) \rangle$

translation of constructors without argument: place the tag into a heap cell

$[\![x]\!]^E = \langle \emptyset, x \rangle$ $\quad [\![con]\!]^E = \langle \emptyset, box(\mathbf{int}, tag) \rangle$ if *con* has no argument

constructors with argument: generate a function that places the tag and the playload onto the heap

$[\![con]\!]^E = \langle \emptyset, gen\text{-}closure\ con_{cl}[] \rangle$ if *con* has an argument
declare *closure* $con_{cl}[](\mathbf{obj}\ payload) : \mathbf{obj} \to con$
declare *constructor* $con(\mathbf{obj}\ payload)$

Figure 6. Translation scheme from Core to Imp.

```
      ite = *(gen-closure binop[])
        (gen-closure ADD_cl[], r/m16(), r16());
    else
      ite = *(gen-closure binop[])
        (gen-closure ADD_cl[], r/m32(), r32());;
    exec ite
```

The resulting code contains the construction of many unnecessary closures such as the invocation `*(gen-closure binop[])` that is equivalent to a direct call to `binop`. The next section presents optimizations to address these issues.

4. Optimizing the Intermediate Representation Imp

This section details various optimizations that bring the code closer to natural C code, that is, C code that might have been written by hand. The challenge lies in removing the functional language artifacts as far as possible. These artifacts mostly relate to the boxing of values. The key transformations to avoid the boxing of functions and values are local simplifications, as detailed next. The transformation of monadic actions and the type-based refinement in Sec. 4.2 and 4.3, respectively, merely transform the program so that further simplifications are possible. Thus, the simplifier is run once after each transformation.

4.1 Simplifying Imp

The simplifier traverses the AST looking for sequences of computations that can be replaced with cheaper or no operations. A dead code elimination pass is run as final pass in order to remove any declarations that are no longer required due to the optimizations. The rules of the simplifier are presented in Fig. 7 and motivated as follows:

- Rules 1 and 2 remove superfluous boxing and unboxing pairs on base types.
- Rule 3 identifies monadic closures that are immediately executed. However, a monadic closure $\lambda\text{S} \to \vec{d}\,;\ \vec{s}\ e$ declares additional variables $\vec{d}$ and executes the statements $\vec{s}$ before returning the value e. Hence, these declarations must be moved to the enclosing block to ensure that they are visible when evaluating $\vec{s}$ and e. The statements $\vec{s}$ must be executed before evaluating e.
- Rule 4 identifies closures that are constructed and then immediately invoked. The pattern corresponds to call `f x` where `f` is defined as a function.
- Rules 5 and 6 inline functions that have been generated during the conversion from Core to Imp, namely functions *select-foo* for a record selector \$*foo* and *update-f1* $\cdots f_n$ for update functions. The rules apply when the functions are applied to a record in which case the select and update expressions from Imp can be used directly.

The rules of the simplifier are rather standard when compiling functional programs [12]. Applying them to the example code of the last section yields:

```
decode() : obj = decodeRes where
  decodeRes = λS → # case omitted
    obj opndsz; obj ite;
    opndsz = $opndsz s;
    if unbox(int,opndsz) then
       ite = binop(gen-closure ADD_cl[],
          r/m16(), r16());
    else
       ite = binop(gen-closure ADD_cl[],
          r/m32(), r32());
    exec ite
```

4.2 Removing Monadic Actions

A major source of inefficiency and illegibility of the code generated so far relates to monadic actions: every Core function whose body consists of a `do`-block returns a closure. One example is the `binop` function that commences as follows:

```
binop(obj cons,obj giveOp1,obj giveOp2) : obj
    = binopRes where binopRes = λS → body
```

Note that, in order to obtain a result from `binop`, it must first be called with its three arguments, then the resulting closure must be run by applying *exec* to it. Since the state of the monad is updated destructively, the translation retains the source code semantics if the sequence in which computations are performed remains the same. In particular, we can rewrite the function `binop` : $(\mathbf{obj}, \mathbf{obj}) \to \text{S} \to \mathbf{obj}$ and all its call-sites so that its type becomes $(\mathbf{obj}, \mathbf{obj}) \to \mathbf{obj}$ without altering when a result is computed. Hence, we replace $\lambda\text{S} \to$ *body* with *body* and we convert each call-site of `binop(...)` to $\lambda\text{S} \to$`binop(...)`. In general, we traverse the program and gather all functions whose top-level expression is a monadic closure $\lambda\text{S} \to \ldots$. If this set contains a function f whose closure is computed anywhere in the program, it has to be removed from the set since a correct transformation would have to transform all invoke-expressions to which the closure of f can flow and, in turn, all other functions from closure computations that flow to this invoke-expression. In the example, all monadic functions can be converted:

```
decode() : obj = decodeRes where # case omitted
  obj opndsz; obj ite;
  opndsz = $opndsz s;
  if unbox(int,opndsz) then
     ite = λS → binop(gen-closure ADD_cl[],
               λS → r/m16(),λS → r16());
  else
     ite = λS → binop(gen-closure ADD_cl[],
               λS → r/m32(),λS → r32());;
  decodeRes = exec ite
```

The converted program cannot be simplified since the monadic abstractions $\lambda\text{S} \to$ are not surrounded by *exec*. Note, however, that the monadic closure that is assigned to *ite* is executed in the last line. We thus apply a transformation by performing a backward-substitution on variables that were generated by the Core to Imp translation. In particular, the assignment `decodeRes = exec ite` is removed and any assignment of the form `ite =` *exp* is replaced by `decodeRes = exec` *exp*. In general, we also propagate the simpler pattern `var1 = var2` (where `var1` was generated by the translation to Imp) backwards and replace `var2 =` *exp* with `var1 =` *exp*. Applying this transformation on the code above and running the simplifier yields:

```
decode() : obj = decodeRes where # case omitted
  obj opndsz; obj ite;
  opndsz = $opndsz s;
  if unbox(int,opndsz) then
     decodeRes = binop(gen-closure ADD_cl[],
            λS → r/m16(),λS → r16());
  else
```

no	rule			remark
1	$box(t, unbox(t, e))$	$\rightsquigarrow$	e	
2	$unbox(t, box(t, e))$	$\rightsquigarrow$	e	
3	$x = exec(\lambda s \to \vec{d};\ \vec{s};\ e)$	$\rightsquigarrow$	$\vec{d}_{\text{old}}\ \vec{d}\ ...\ \vec{s};\ x = e$	insert declarations $\vec{d}$ in surrounding block
4	$*(gen\text{-}closure\ f_{cl}[\vec{a}])(\vec{a}')$	$\rightsquigarrow$	$f(\vec{a}\ \vec{a}')$	f_{cl} is declared as $closure\ f_{cl}[\vec{d}](\vec{d}') : t_r \to f$
5	$select\text{-}f(r)$	$\rightsquigarrow$	$\$f(r)$	$select\text{-}f$ was generated for the selector $\$f$
6	$update\text{-}f_1 \cdots f_n(e_1 \ldots e_n\ r)$	$\rightsquigarrow$	$@\{f_1 = e_1 \ldots f_n = e_n\}\,(r)$	$update\text{-}f_1 \cdots f_n$ was generated for the record update function $@\{f_1 = e_1 \ldots f_n = e_n\}$

Figure 7. Rules of the Simplifier.

```
8       decodeRes = binop(gen-closure ADD_cl[],
9               λs → r/m32(),λs → r32());;
```

Finally, we perform a pass that converts all expressions $\lambda s \to body$ into a top-level function containing *body* that takes the free variable $x_1, \ldots x_n$ of *body* as arguments. The λ-expression is thus replaced by `gen-closure f[`$x_1, \ldots x_n$`]`. Analogously, *exec e* is turned into a function invocation $*e()$. A better translation is possible for patterns of the form $\lambda s \to f()$ since replacing it by the closure of a top-level function that calls $f()$ is equivalent to replacing $\lambda s \to f()$ by `gen-closure f[]`. Thus, in the example, the monadic closure $\lambda s \to$ `r/m16()` is translated to `gen-closure r/m16[]`.

4.3 Unboxing by Type Inference

By default, all variables are pointers to the heap, thus requiring heap-allocating the result of each computation which is slow and produces hard-to-read C code. Our central and – as far as we know – novel optimization is a type inference that determines which data can be stored in variables.

The type universe of Imp is shown in Fig. 8 a). Here, the special type **void** represents the empty set of program values whereas **obj** represents all possible program values. Other types are **bits**$[n]$ for vectors of n bits, **vec** for bit vectors whose size is not statically known and whose size is therefore tracked at runtime, monadic actions $\mathbf{M}\ r$ with result r, boxed types **box**$[t]$ where t is not a monadic action and record types. Figure 9 presents typing rules that characterize Imp programs that have monomorphic typings which are exactly those programs whose variables have a fixed type in all executions and which therefore can be unboxed. Most Imp programs are not well-typed under these rules, in these cases, our inference will over-approximate this typing.

The top three rules in Fig. 9 specify how the top-level declarations are represented in the environment Γ. Rules for statements and for expressions follow. A specialty of our types is the flag f in a function type $(t_1 \ldots t_n) \xrightarrow{f} t$ that is *true* if $k > 0$ in rule (CLO), that is, if the closure contains environment variables. Such non-trivial closures always have boxed function types.

Another non-standard aspect are the types of records that may be either flexible (later represented by a linked list of field/value pairs) or fixed (later represented by a C `struct`). Any record r to which *update* r is applied is flexible, its type is **obj** and the type of the fields is given in a global map Φ (rules (UPD), (REC-GLOB) and (SEL-GLOB)). Fixed records have the type $\{f_1 : t_1, \ldots f_n : t_n, all\}$ where the flag *all* indicates whether all fields of the record are known. For instance, the set of fields in a record $\{f_1 = 1, f_2 = 2\} = \{f_1 : \mathbf{int}, f_2 : \mathbf{int}, true\}$ is always known whereas the set in a selector type $\$f_1 : (\{f_1 : t, false\}) \to t$ is not known which is reflected by the *all* flag. The rule (SUB) allows to promote records with a known set of fields to one where fields are omitted, thereby allowing $\$f_1$ to be applied to $\{f_1 = 1, f_2 = 2\}$. The remaining rules are standard.

In order to infer a typing that is sound with respect to these rules, we define the union of two types $t_1 \sqcup t_2$ in Fig. 8b). The shown rules are to be read from top to bottom, thus, **bits**$[5] \sqcup$ **bits**$[5] =$ **bits**$[5]$ due to the third rule. The union of two fixed records is defined in Fig. 8c). The first rule applies if all fields are known in one of the records (say, the right one: $e_r = true$) and the other record contains extra fields ($n \neq k$). Returning **obj** implies that the record can only be represented as a flexible record. For example, consider the `insn` type in Fig. 1 and the expression `case inst of ADD arg2 -> eval arg2 | INC arg1 -> eval arg1`. Since `arg1` and `arg2` are both passed to `eval`, a common type must be computed. But $\{opnd1 : \mathbf{obj}, opnd2 : \mathbf{obj}, true\} \sqcup \{opnd1 : \mathbf{obj}, true\} = \mathbf{obj}$ since $e_r = true \wedge 2 = n \neq k = 1$. Hence, the argument to `eval` and to `ADD` and `INC` must be a flexible record.

Algorithmically, we replace the **void** type in the universe of types **Type** with a set of type variables and associate a different variable with each record operation and all positions in the program where the grammar in Fig. 6 specifies **Type**. Based on the typing rules, we equate these type variables with the stipulated types and apply the union operation $\sqcup$ if two types differ. This inference can be implemented very efficiently using a union-find data structure where an equivalence class representative holds the type of the type variables in that class.

Note that our type inference is non-standard: a fresh type variable corresponds to the **void** type and equating it with other types applies $\sqcup$ which encodes anti-unification. In the worst case, the most general type **obj** is inferred. In contrast, standard type inference performs unification and a fresh type variable corresponds to **obj**, the most general type. In the worst case, the least general type **void** is inferred, indicating a type error since no program values exist in this type. Thus, while standard type inference infers the most general type, ours infers the most specific type, thereby solving the problem of inferring a type that is sufficiently specific to unbox the arguments of the `cmp` function in Sec 1.

Given the inferred types, Fig. 10 shows the rules used to re-write various expressions in the program. The rewriting of function return values and arguments are handled analogously to writing/reading rules. The *box*/*unbox* constructors also show when a variable contains a bit-vector **bits**$[c]$ of fixed size c in which case the bit-string is stored as **int**.

Finally, we infer a fixed record for the monadic state if all update actions are of the form *update* $@\{\ldots\}$ and all query actions are of the form *query* $\$f$. In this case the record does not escape and can be updated in-situ. An update rule analogous to (SEL) infers the set of fields and we translate *update* $@\{f = 42\}$ to `state.f=42` where `state` contains

a)

$$
\begin{array}{lcl}
\textbf{Type} & ::= & \textbf{void} \mid \textbf{obj} \mid \textbf{int} \\
& \mid & \textbf{bits}[\underline{num}] \mid \textbf{vec} \mid \textbf{str} \\
& \mid & (\textbf{Type}^*) \xrightarrow{\textbf{Flag}} \textbf{Type} \\
& \mid & \textbf{box}[\textbf{Type}] \mid \textbf{M}\ \textbf{Type} \\
& \mid & \{\,(\underline{field} : \textbf{Type}\ \underline{,})^*\ \textbf{Flag}\,\} \\
\textbf{Flag} & ::= & \underline{true} \mid \underline{false}
\end{array}
$$

b)

$$
\begin{array}{lcl}
\textbf{void} \sqcup t & = & t \\
t \sqcup \textbf{void} & = & t \\
t \sqcup t & = & t \\
\textbf{bits}[c_1] \sqcup \textbf{bits}[c_2] & = & \textbf{vec} \\
\textbf{box}[t_1] \sqcup \textbf{box}[t_2] & = & \textbf{box}[t_1 \sqcup t_2] \\
\textbf{M}\ t_1 \sqcup \textbf{M}\ t_2 & = & \textbf{M}\ (t_1 \sqcup t_2) \\
(t_1, \ldots t_n) \xrightarrow{c} r \sqcup (t'_1, \ldots t'_n) \xrightarrow{c'} r' & = & (t_1 \sqcup t'_1, \ldots t_n \sqcup t'_n) \xrightarrow{c \vee c'} r \sqcup r' \\
\{f_l, e_l\} \sqcup \{f_r, e_r\} & = & \mathit{mergeRec}(f_l, e_l, f_r, e_r) \\
t_1 \sqcup t_2 & = & \textbf{obj}
\end{array}
$$

c)

$$
\mathit{mergeRec}\left(\begin{array}{l} f^c_1 : t^l_1, \ldots f^c_k : t^l_k, f^l_{k+1} : t^l_{k+1}, \ldots f^l_n : t^l_n, e_l, \\ f^c_1 : t^r_1, \ldots f^c_k : t^r_k, f^r_{k+1} : t^r_{k+1}, \ldots f^r_m : t^r_m, e_r \end{array}\right) =
$$

$$
\begin{cases}
\textbf{obj} & \text{if } e_r \wedge n \neq k \vee e_l \wedge m \neq k \\
\{f^c_1 : t^l_1 \sqcup t^r_1, \ldots f^c_k : t^l_k \sqcup t^r_k, f^l_{k+1} : t^l_{k+1}, \ldots f^l_n : t^l_n, f^r_{k+1} : t^r_{k+1}, \ldots f^r_m : t^r_m, e_l \vee e_r\} & \text{otherwise}
\end{cases}
$$

Figure 8. Definition of Types, their union and merging of record types.

the C `struct` of the monadic state. This is exemplified by lines 33–35 of Fig. 3.

5. Implementation

We designed our *generic decoder specification language* (GDSL[1]) with the goal to write concise x86 instruction decoders, semantic translators to an intermediate representation called RReil [13] and RReil optimizers. The decoder handles all 897 Intel instructions and has been validated against the Intel XED library [4]. We offer semantic translations for about half the instructions. The optimizations are crucial to reduce the size of the resulting code.

Figure 11 illustrates how a GDSL library is synthesized. In particular, a generic runtime template is specialized for the input GDSL program by adding declarations for the monadic state, constructors, and heap allocation functions. This bespoke runtime is combined with the generated code and compiled to a library. A host C program can then link against this library.

Our C backend adds a pointer to the runtime environment `s` as the first argument to every emitted function. This runtime environment also contains the monadic state in `s->state`, thereby alleviating the need for any global state and, thus, making the library thread-safe - this is important since GDSL does not provide means for concurrency itself and, thus, concurrency has to be implemented by using multiple GDSL instances. As hinted at in Fig. 11, the host program obtains an initial runtime environment by calling `init()`. It is then at liberty to call and use the result of any GDSL function that returns basic C types and `struct`s. In order to transfer algebraic data types, such as ASTs, to C, a structural induction can be programmed in the DSL that calls a different function for each constructor. Passing these functions as a record generates a traversal function that takes a C `struct` containing well-typed pointers to C functions. Thus, marshaling data between the DSL and C is very simple and no extra tools are needed.

Once the results are extracted, the host program may reset the GDSL heap through the `reset_heap()` function. The function frees all heap space except for the first page which yields a slight performance advantage when running many small DSL functions. GDSL uses fast bumper pointer allocation [9] within each page and increases the heap size by one four kilobyte pages when running out of memory.

[1] The toolkit is available at `http://code.google.com/p/gdsl-toolkit`

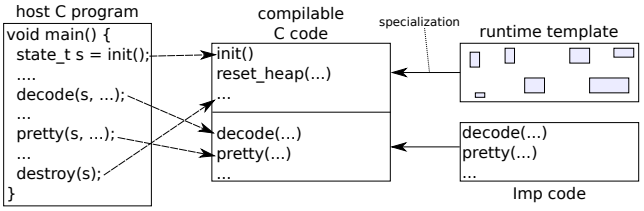

Figure 11. GDSL program assembly.

The GDSL language requires the programmer to specify which functions to export. All non-exported functions are declared static and are aggressively optimized by the C compiler. Indeed, by annotating the runtime function that allocates a new page so that it is not inlined, the size of the executable decreases by nearly one third. Modern C compilers turn tail-recursive function calls into jumps, thereby allowing recursive loops written in the DSL to run without any overhead over loops written in C.

6. Experimental Evaluation

We evaluated our implementation regarding the following three aspects: effectiveness of the optimizations, comparison of the performance to hand-written code, and heap consumption for various task sizes. We assess the optimizations by benchmarking the decoding and pretty printing of the 11Mb `clang` binary. Figure 12 shows the results for different GDSL compiler optimization configurations. We use the term *fixed records* when emitting C `struct`s whenever possible. We call records unboxed if the C `struct`s are passed by value rather than allocated on the heap. In contrast, compilation without fixed records exclusively relies on lists of field/value pairs which are always heap-allocated. The table shows the difference to the optimal case where type-based refinement has been applied, records are fixed if possible and all fixed records are unboxed. The second column shows the size of the generated C code. Thereafter, the binary code size of the decoder and the program runtime are displayed both with and without C compiler optimizations (-O0 and -O2). Finally, the last columns contain the average per-instruction and the maximum heap residency of the decoder.

$$\frac{\Gamma(f) = (t_1 \dots t_n) \overset{false}{\to} t_r \qquad \Gamma(a_i) = t_i \qquad \Gamma(r) = t_r \qquad \Gamma \vdash b : t_r}{\Gamma \vdash f(t_1 a_1 \dots t_n a_n) : t_r = r \ \textit{where}\ b}\ (\textsc{fun}) \qquad \frac{\Gamma(id) = (t) \overset{false}{\to} \mathbf{obj}}{\Gamma \vdash \textit{constructor}\ id(t\ a)}\ (\textsc{con})$$

$$\frac{\Gamma(cid) = (t_1 \dots t_k) \overset{false}{\to} (t_{k+1}, \dots t_n) \overset{k>0}{\to} t_r \qquad \Gamma(f) = (t_1 \dots t_n) \overset{false}{\to} t_r}{\Gamma \vdash \textit{closure}\ cid[t_1 a_1 \dots t_k a_k](t_{k+1} a_{k+1} \dots t_n a_n) : t_r \to f}\ (\textsc{clo})$$

$$\frac{\Gamma(x) = t \qquad \Gamma \vdash e : t}{\Gamma \vdash x = e;}\ (\textsc{ass}) \qquad \frac{\Gamma \vdash e : \mathbf{int} \qquad \Gamma \vdash b_t \qquad \Gamma \vdash b_e}{\Gamma \vdash \textit{if}\ e\ \textit{then}\ b_t\ \textit{else}\ b_e;}\ (\textsc{if})$$

$$\frac{\Gamma \vdash e : \mathbf{int} \qquad \Gamma \vdash b_i}{\Gamma \vdash \textit{case}\ e\ \textit{of}\ p_1 : b_1; \dots p_n : b_n;}\ (\textsc{case}) \qquad \frac{\Gamma[x_1 \mapsto t_1, \dots x_n \mapsto t_n] \vdash s_i}{\Gamma \vdash t_1\ x_1 \dots t_n\ x_n; s_1 \dots s_m}\ (\textsc{block})$$

$$\Gamma \vdash id : \Gamma(id)\ (\textsc{var}) \qquad \Gamma \vdash 42 : \mathbf{int}\ (\textsc{int}) \qquad \Gamma \vdash \text{"}string\text{"} : \mathbf{str}\ (\textsc{str}) \qquad \Gamma \vdash \text{'}b_1 \dots b_n\text{'} : \mathbf{bits}[n]\ (\textsc{vec})$$

$$\frac{\Gamma \vdash e : (t_1 \dots t_n) \overset{false}{\to} t_r \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash e(e_1 \dots e_n) : t_r}\ (\textsc{call}) \qquad \frac{\Gamma \vdash e_i : t_i}{\Gamma \vdash \{f_1 = e_1 \dots f_n = e_n\} : \{f_1 : t_1 \dots f_n : t_n, true\}}\ (\textsc{rec})$$

$$\frac{\Gamma \vdash e : \mathbf{box}[(t_1 \dots t_n) \overset{non\text{-}triv}{\to} t_r] \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash *e(e_1 \dots e_n) : t_r}\ (\textsc{inv})\ \text{where}\ non\text{-}triv \in \{true, false\} \qquad \frac{\Gamma \vdash r : \{f : t, false\}}{\Gamma \vdash \$f\ r : t}\ (\textsc{sel})$$

$$\frac{\Gamma \vdash e_i : t_i \qquad \Phi(f_i) = t_i}{\Gamma \vdash \{f_1 = e_1 \dots f_n = e_n\} : \mathbf{obj}}\ (\textsc{rec-glob}) \qquad \frac{\Gamma \vdash r : \mathbf{obj} \qquad \Phi(f) = t}{\Gamma \vdash \$f\ r : t}\ (\textsc{sel-glob})$$

$$\frac{\Gamma \vdash e_i : t_i \qquad \Gamma \vdash r : \mathbf{obj} \qquad \Phi(f_i) = t_i}{\Gamma \vdash @\{f_1 = e_1 \dots f_n = e_n\}\ r : \mathbf{obj}}\ (\textsc{upd}) \qquad \frac{\Gamma \vdash \{f_1 : t_1 \dots f_k : t_k \dots f_n : t_n, true\}}{\Gamma \vdash \{f_1 : t_1 \dots f_k : t_k, false\}}\ (\textsc{sub}) \qquad \frac{\Gamma \vdash e : \mathbf{M}\ t}{\Gamma \vdash \textit{exec}\ e : t}\ (\textsc{exec})$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \textit{box}(t, e) : \mathbf{box}[t]}\ (\textsc{box}) \qquad \frac{\Gamma \vdash e : \mathbf{box}[t]}{\Gamma \vdash \textit{unbox}(t, e) : t}\ (\textsc{unbox})$$

$$\frac{\Gamma \vdash e_i : t_i \qquad \Gamma(cid) = (t_1 \dots t_k) \overset{false}{\to} (t_{k+1} \dots t_n) \overset{non\text{-}triv}{\to} t_r}{\Gamma \vdash \textit{gen-closure}\ cid[e_1 \dots e_k] : \mathbf{box}[(t_{k+1} \dots t_n) \overset{non\text{-}triv}{\to} t_r]}\ (\textsc{gen-clo}) \qquad \frac{\Gamma \vdash b : t_b \qquad \Gamma \vdash e : t_r}{\Gamma \vdash \lambda_S \to b\ e : \mathbf{M}\ t_r}\ (\textsc{do})$$

Figure 9. Typing rules that characterize programs on which unboxing can be applied.

rule	inferred	new type	original	new
writing x	$\mathbf{box}[\mathbf{int}]$	$\mathbf{int}$	$x = e$	$x = \textit{unbox}(\mathbf{int}, e)$
reading x	$\mathbf{box}[\mathbf{int}]$	$\mathbf{int}$	x	$\textit{box}(\mathbf{int}, x)$
writing x	$\mathbf{box}[\mathbf{vec}]$	$\mathbf{vec}$	$x = e$	$x = \textit{unbox}(\mathbf{vec}, e)$
reading x	$\mathbf{box}[\mathbf{vec}]$	$\mathbf{vec}$	x	$\textit{box}(\mathbf{vec}, x)$
writing x	$\mathbf{box}[\mathbf{bits}[c]]$	$\mathbf{int}$	$x = e$	$x = \textit{unbox}(\ \mathbf{bits}[c], \textit{unbox}(\mathbf{vec}, e))$
reading x	$\mathbf{box}[\mathbf{bits}[c]]$	$\mathbf{int}$	x	$\textit{box}(\mathbf{vec}, \textit{box}(\mathbf{bits}[c], x))$
closure $cid \to f$	$\mathbf{box}[(\vec{t}) \overset{false}{\to} t_r]$	$(\vec{t}) \overset{false}{\to} t_r$	$\textit{gen-closure}\ cid[]$	f
invoke e	$\mathbf{box}[(\vec{t}) \overset{false}{\to} t_r]$	$(\vec{t}) \overset{false}{\to} t_r$	$*e(a_1, \dots a_n)$	$e(a_1, \dots a_n)$

Figure 10. Unboxing rules.

Figure 13 compares our decoder to Intel's XED decoder [4] using the same binary input as above. In previous work, we have observed XED to be the fastest freely available instruction decoder that is also correct [14]. Since XED is distributed in compiled form, we cannot assess what changes were made between versions 2.11 and 2.12 besides them being compiled by different gcc compiler versions. Still an increase in speed by nearly 30% suggests that the library has been manually tuned for speed. Given that all optimizations that our GDSL program has encountered lie in the application-agnostic optimization passes described in this paper, we believe that this highlights the merit of using pure functional programs based on a state monad as core for a DSL.

Finally, Fig. 14 presents measurements for different GDSL programs. The test cases are ordered by task com-

optimization options	lines of code	with -O0 exe size	with -O0 time	with -O2 exe size	with -O2 time	avg. heap residency	max. heap residency
all optimizations	36k	511kb	3.6s	295kb	1.1s	1.0kb	2.5kb
all w/o unboxed records	36k	499kb	4.0s	284kb	1.2s	1.1kb	3.14kb
all w/o type refinement	41k	855kb	5.4s	719kb	1.5s	1.9kb	6.2kb
all w/o fixed records	37k	605kb	9.9s	388kb	3.7s	4.6kb	10.9kb
all w/o fixed records w/o type refinement	42k	945kb	11.4s	764kb	3.8s	5.3kb	14.4kb

Figure 12. Decoding performance depending on the GDSL compiler optimization level.

decoder	time	dec. insn.	exe size
XED 2.12	1.2s	2667248	1344kb
XED 2.11	1.7s	2667248	1024kb
GDSL opt.	1.1s	2667248	295kb

Figure 13. Decoding performance of XED from the Intel Pin toolkit.

plexity; we used the same test input as above. The table starts by recapitulating the performance of the decoder. The decoder processes the binary instructions independently, that is, the heap is reset after the decoding of each instruction. Next, measurements for the semantic translation of single instructions are shown. Again, instructions are handled independently, but the heap can only be reset after the semantic translation of a decoded instruction completes. The third line of the tables provides measurements for our liveness analysis which processes the binary input basic-block-wise. In order to perform the analysis, the translation of the whole basic block needs to be kept in memory and the heap can only be reset after finishing that block. Finally, the last line presents measurements for an enhanced liveness analysis that does not only consider one basic block, but also its successors (if they can be determined). Here, the data for up to three basic blocks needs to be stored in memory. The high maximal memory usage of 67Mb indicate that there is a single basic block that is rather larger. In production quality applications, the task size run as DSL program should be artificially limited in size in order to prevent an out-of-heap situation. In our case, it would be enough to split up basic blocks once they have reached a certain sizes.

The open-source GDSL compiler and complete examples of the generated C code evaluated in Fig. 12 are available at `http://code.google.com/p/gdsl-toolkit/`.

7. Related Work

A common perception of domain-specific languages is that they should be "small" and not Turing-complete [7] since they otherwise encompass more than the domain-specific aspect of the problem. This work proposes that a general functional language as carrier is an effective approach to symbolic computations. Since it is seamlessly embedded into C, as soon as logic has to be encoded for which a pure functional language is unsuitable, it can be easily implemented in C.

Our translation scheme can furthermore be characterized as a "shallow embedding", that is, an implementation that translates the DSL program into the native operations of the target language. In contrast, a "deep embedding" is an implementation in which the DSL program is evaluated using an interpreter written in the target language [6]. In practice, there is a continuum between these extremes and, indeed, our type-based optimization transforms the DSL program to be more shallow: as an example, a non-optimized program represents every record as a linked list of field/value pairs which is not a common way to represent data structures in C. The optimization of turning most of the records into C `struct`s makes the DSL program more C-like and, hence, more shallow.

While many DSLs focus on making programming simpler and safer, the FFTW library for computing discrete Fourier transforms uses a DSL specifically to obtain programs that are more efficient than hand-written C programs [8]. The underlying principle is to express the transform as a directed acyclic graph of codelets that implement building blocks and to optimize this graph by rewriting. Another example is the Pan library for image manipulation [5] which embeds the DSL into the general purpose language Haskell. The DSL constructs are Haskell functions that generate an abstract syntax tree which is then optimized using inlining and common-subexpression elimination. The resulting code is emitted as C code and compiled by a C compiler. In both approaches, the goal of obtaining good performance sacrifices the structure of the input program during translation, so that finding bugs in the emitted program is difficult for the user of the DSL. An even more ambitious way of optimizing a functional DSL is to use standard compiler techniques such as a translation to continuation passing style (CPS). Our initial backend was based on the CPS transformation and optimization by Kennedy [10]. We found that removing the bind function that concatenates two monadic actions was crucial to obtain code with few closure allocations. Unfortunately, this optimization required a somewhat aggressive inlining (using β-reduction). Controlling the inlining turned out to be a major difficulty and even in the best setting, a translation of the x86 decoder alone resulted in 13MB of C code which becomes difficult to compile and very hard to debug. Moreover, the resulting code had about 1/3 of the performance of our current backend. The presented translation scheme therefore lies at a sweet-spot in that it creates readable C code that has a small executable footprint while achieving the performance of hand-written C/C++ code.

The idea of performing a task on a fresh heap and discarding the heap upon completion is a form of region-based memory management [9]. It finds widespread use in the implementation of plug-in modules for the Apache web-server. In some applications, the DSL program might allocate a lot of memory and a garbage collector might be useful. However, a generic garbage collector for our DSL requires that data on the heap is tagged, which may reduce the performance of the generated code. Instead, a built-in garbage collection primitive could be added that allows the programmer to specify which data to keep and ensures that all other identifiers can no longer be accessed. In this case, the compiler

GDSL program	time	heap residency avg.	max.	allocation rate
x86 decoder + printing	1.2s	1.0kb	2.5kb	2090Mb/s
x86 decoder + translator + printing	8.2s	7.7kb	66kb	2459Mb/s
x86 decoder + translator + liveness	92s	206kb	67Mb	1305Mb/s
x86 decoder + translator + lookahead liveness	241s	497kb	67Mb	1203Mb/s

Figure 14. GDSL program performance using all optimizations

would generate a bespoke copying function for the data to keep based on the types. Note that these types must be monomorphic since copying polymorphic types requires the data on the heap to be tagged in order to determine the size of the heap object. An interesting case is to apply the garbage collection primitive only to the monadic state. Such a setup would enable the use of high-level, functional DSLs in small embedded control systems that repeatedly execute an infinite loop. The DSL would require that the state of the system is stored inside the monad which is then the only data that remains alive between loop iterations.

Type-based unboxing of heap values has been investigated by Leroy in the context of the OCaml compiler [12]. His work was generalized to also cope with the module system of ML and, thus, separate compilation [15]. The latter work observes that many functions are used with a type that is more monomorphic than their inferred type as illustrated by the `cmp` function in the introduction. Since a polymorphic argument to a function requires that it is boxed, Bjørner proposes to specialize a function type as much as possible in order to perform more unboxing [2]. Our approach achieves the same effect by performing a monomorphic type inference and to use the special type **obj** to represent a type that is not monomorphic. Moreover, any constructor that is only used to store specific types will have its payload stored unboxed. Thus, a polymorphic list that is only used with integers will store the integer directly in each list cell. On the downside, compiling DSL modules separately requires that the monomorphic type information can be propagated between the different modules in order to propagate the type requirements between all call sites and functions. The infrastructure for communicating types between separate modules is always built into the type checker/inference of the compiler front-end but is unlikely to be available to the backend. Thus, our monomorphic type inference is easiest to apply as a whole-program analysis that requires all DSL modules at once.

8. Conclusion

We have presented a translation scheme for a purely functional language that provides a built-in state monad. The goal is to perform a structure-preserving translation that enables the user to easily relate the emitted code with the DSL program and thereby allowing for simple debugging and profiling using the emitted code. To this end, several simple transformations were presented that replace concepts from the input DSL (such as boxed values, closures, and curried functions) into concepts used in C programs (such as `struct`s and function pointers). The key insight is that even larger DSL programs are often simple enough to optimize most closure and boxing operations away, thereby yielding an imperative program that resembles hand-written C code. Due to this resemblance, many of the optimizations found in off-the-shelf C compilers are applicable to the generated code, thus yielding a highly efficient executable.

Acknowledgments

This work was supported by the DFG Emmy Noether project SI1579.

References

[1] L. Bettini. Implementing Java-like languages in Xtext with Xsemantics. In S. Y. Shin and J. C. Maldonado, editors, *Symposium on Applied Computing*, pages 1559–1564, Coimbra, Portugal, Mar. 2013. ACM.

[2] N. S. Bjørner. Minimal Typing Derivations. In *Workshop on ML and its Applications*, pages 120–126. ACM, 1994.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[4] I. Corp. Pin - a dynamic binary instrumentation tool. `http://www.pintool.org`. URL `http://www.pintool.org`.

[5] C. Elliott, S. Finne, and O. de Moor. Compiling Embedded Languages. *Journal of Functional Programming*, 13(2), 2003.

[6] M. Erwig and E. Walkingshaw. *Semantics-Driven DSL Design*, chapter 3, pages 56–80. IGI Global, Oct. 2013. .

[7] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321712943, 9780321712943.

[8] M. Frigo. A fast Fourier transform compiler. In *Programming Language Design and Implementation*, pages 169–180, Atlanta, Georgia, USA, 1999. ACM. ISBN 1-58113-094-5. .

[9] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Aug. 2012. ISBN 978-1420082791.

[10] A. Kennedy. Compiling with Continuations, Continued. In *International Conference on Functional Programming*, pages 177–190, Freiburg, Germany, 2007. ACM. .

[11] J. Kranz, A. Sepp, and A. Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In C. Shan, editor, *Asian Symposium on Programming Languages and Systems*, Melbourne, Australia, Dec. 2013. Springer.

[12] X. Leroy. Unboxed objects and polymorphic typing. In *Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[13] A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M.Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, Oct. 2011. IEEE.

[14] A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. In *Tools for Automatic Program Analysis*, ENTCS, Deauville, France, Sept. 2012. Springer. URL `https://bitbucket.org/mb0/gdsl`.

[15] Z. Shao and A. W. Appel. A Type-based Compiler for Standard ML. In *Programming Language Design and Implementation*, pages 116–129, La Jolla, California, USA, June 1995.