Higher-Order Specifications for Deductive Synthesis of Programs with Pointers

David Young* 
University of Kansas, USA

Ziyi Yang* 
National University of Singapore, Singapore

Ilya Sergey 
National University of Singapore, Singapore

Alex Potanin 
Australian National University, Australia

Abstract

Synthetic Separation Logic (SSL) is a formalism that powers SuSLik, the state-of-the-art approach for the deductive synthesis of provably-correct programs in C-like languages that manipulate heap-based linked data structures. Despite its expressivity, SSL suffers from two shortcomings that hinder its utility. First, its main specification component, inductive predicates, only admits *first-order* definitions of data structure shapes, which leads to the proliferation of "boiler-plate" predicates for specifying common patterns. Second, SSL requires *concrete* definitions of data structures to synthesise programs that manipulate them, which results in the need to change a specification for a synthesis task every time changes are introduced into the layout of the involved structures.

We propose to significantly lift the level of abstraction used in writing Separation Logic specifications for synthesis—both simplifying the approach and making the specifications more usable and easy to read and follow. We avoid the need to repetitively re-state low-level representation details throughout the specifications—allowing the reuse of different implementations of the same data structure by abstracting away the details of a specific layout used in memory. Our novel *high-level front-end language* called Pika significantly improves the expressiveness of SuSLik.

We implemented a layout-agnostic synthesiser from Pika to SuSLik enabling push-button synthesis of C programs with in-place memory updates, along with the accompanying full proofs that they meet Separation Logic-style specifications, from high-level specifications that resemble ordinary functional programs. Our experiments show that our tool can produce C code that is comparable in its performance characteristics and is sometimes faster than Haskell.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Automatic programming; Software and its engineering → Compilers

Keywords and phrases Program Synthesis, Separation Logic, Functional Programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.34

Supplementary Material An artefact approved by ECOOP 2024 Artefact Evaluation Committee
Software (Source Code): https://github.com/roboguy13/PikaC
Software (Docker image for artefact evaluation): https://zenodo.org/records/10558356

Acknowledgements We thank the anonymous ECOOP 2024 PC and AEC reviewers for their constructive and insightful comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001 and MoE Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems".

* Contributed equally to this work.

1 Introduction

Recent advances in program synthesis have allowed programmers to concentrate on stating precise specifications—leaving the job of generating provably correct and efficient imperative code to the synthesiser, such as SuSLik [6, 9, 13]. Such specifications are usually expressed using (Synthetic) Separation Logic [7, 10] that while hugely successful in verifying properties of pointer-manipulating programs remains out of reach to many mainstream developers. As the programs grow in complexity, such SSL specifications can become exceedingly verbose and complex—making the job of specification writer especially error-prone and defeating the purpose of a *usable proof automation toolchain*.

The power of Separation Logic (SL) specifications for the tasks of both verification and synthesis, is its mechanism of *inductive predicates* that concisely capture the shape of possibly recursive pointer-based tree-like data structures, determining both induction schemes for verification and the shape of recursion for the synthesis tasks of the programs that manipulate such data structures [10]. The surprising ability of SL specifications to capture precisely the logic of a desired program to be synthesised in the logical assertions comes at the price of the involved inductive predicates being (a) *first-order* and (b) somewhat *low-level*, with both these aspects posing limitations to the usability of SL-based program synthesis.

The first-order nature of the predicates means, for instance, that synthesis tasks that involve several data structures with very similar heap layouts would require to use *different* predicate definitions. As a specific example, consider a task synthesising two functions, `f` and `g`. Both `f` and `g` take as an argument a pointer to a linked list of integers; `f` increments all elements of the list by one, while `g` multiplies all its elements by two. To specify these two tasks, the state-of-the-art tools for program synthesis based on SL specifications require the user to provide, in addition to the pre-/postconditions, *three* different inductive predicates: one for an arbitrary list, another for a list that carries a known payload, with each element incremented by one, and the final capturing the multiplication of each element by two.

The second aspect, *i.e.*, the low-level nature of the SL inductive predicates used for synthesis, shows up when we try to *rewrite* an already specified synthesis task for a data structure with a slightly different layout. As an example, imagine defining the task of concatenating two lists. It is natural to expect that the specification will look very similar for both singly and doubly linked lists. Yet, since those are two different structures, with two different layouts, the user would require to supply two different task specifications.

A seasoned programmer would immediately notice that both issues (a) and (b) very much resemble the struggle that one faces when programming in a language that does not provide certain abstractions, which are nowadays mostly taken for granted: *higher-order functions* and *abstract data types*. Our first example could be streamlined should the SL-based specification language offer a way to define a function similar to `List.map`, available in all popular functional programming languages, so it could be used to concisely express the two scenarios of manipulating with the payload of a list's elements. The second example would benefit from the ability to specify concatenation of *abstract* lists, while separately handling manipulations with single- or doubly-linked lists in terms of their memory *layouts*.

Key Ideas

The two challenges faced by the SSL specification language for the synthesis of heap-manipulating programs—the need for higher-order functions and abstract data types—have provided the primary motivation for this work.

As a solution, we developed Pika: a high-level front-end language and specification

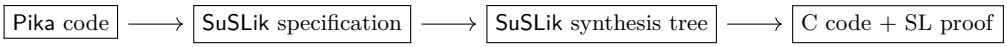


Figure 1 Pika translation pipeline

translation framework built on to of the state-of-the-art SL-based program synthesiser SuSLik.[1] SuSLik is based on a variant of separation logic called *synthetic separation logic* or SSL [9]. Pika has a syntax similar to popular functional programming languages and features specification-level higher-order functions on Algebraic Data Types (ADTs). In addition to being more succinct in comparison to Synthetic Separation Logic (SSL), the specification formalism of SuSLik, Pika also addresses its reusability issues outlined above. First, the use of specification-level higher-order functions allows the user to abstract over the specific properties of the payloads of the heap-based data structures, thus, generalising existing inductive predicates for so they could be employed in a wider range of synthesis tasks. Second, by manipulating ADTs, the synthesis specifications do not need to deal with the specific memory layouts of the data structures. This separates the specification of the tasks that operate on those data structures from the low-level details of their memory representations.

The Pika pipeline is depicted in Figure 1. As our goal was to extend the expressivity of SuSLik by giving it a high-level specification language while retaining its meta-theoretical guarantees (*i.e.*, the certifiable correctness of the programs it synthesises), we had to overcome several technical obstacles when designing Pika and implementing it on top of SuSLik. First, we had to formally define the operational semantics of Pika and its translation to SSL-based specifications of SuSLik, defining the corresponding soundness result, relating the behaviour of programs eventually synthesised to that of their high-level counterparts (Section 3). Second, to support the higher-order specifications of Pika, we introduced conservative extensions to SuSLik's specification language as well as to its deductive synthesis rules, to make it capable of handling pre-/postconditions with first-class functions operating on the payload of heap-stored data types (Section 4).

Pika as a Programming Language

Given the close similarity of Pika, our new specification language, to general-purpose functional programming languages, such as Haskell, it is natural to wonder whether it's possible to leverage its underlying synthesis pipeline as a way to produce efficient imperative programs in a language such as C from equivalent high-level functional programs. In other words, if we decide to use a combination Pika + SuSLik as a *compiler*, would it be a viable replacement to many-decades old tools such as Glasgow Haskell Compiler (GHC) [3], for producing efficient runnable code? To answer this question, we have conducted evaluation on several list- and tree-manipulating benchmarks, comparing the performance of verified C code, emitted by SuSLik from Pika specifications, to that of executables produced for equivalent tasks by GHC.

Our preliminary results are encouraging: thanks to avoiding unnecessary allocation and using destructive heap updates whenever possible, the C code synthesised by Pika + SuSLik outperforms the GHC output (with compiler optimisations flags turned) in the majority of list-manipulating benchmarks we've tried; our synthesis tool also produces strictly more performant C code for tree-manipulating benchmarks when compared to the corresponding

[1] Both susliks and pikas are Central Asian mammals. Pikas might look similar to susliks, but are more nimble and have longer life expectancy.

Haskell programs, compiled by GHC without or with optimisations (Section 5). In particular, we specifically observe this when we compare C code synthesised by Pika + SuSLik with no C compiler optimisation flags to GHC compiled code with no GHC compiler flags.

Contributions

In this work, we make the following contributions:

- We address the expressivity limitations of SSL, the specification formalism of the state-of-the-art deductive synthesis tool SuSLik by developing Pika—a high-level specification language with higher-order functions and abstract data types.
- We formally define the operational semantics of Pika and prove the soundness of translation from Pika to pre-/postconditions in SSL.
- We develop an extension to SuSLik's specification and synthesis mechanism that enables translation from Pika specifications featuring first-class functions.
- We observe that the synthesis tool resulting from the combination Pika + SuSLik enables certified memory-layout-agnostic compilation from a functional specification to C code.
- We report on the evaluation of the Pika regarding its expressiveness and performance. In particular, we show that the C code it produces frequently outperforms equivalent Haskell programs compiled by GHC.

2 Overview

2.1 Background

The Pika language is translated into SuSLik [9], which is a program synthesis tool that uses Synthetic Separation Logic (SSL)-a variant of Hoare-style [4] Separation Logic (SL) [7].

A synthesis task specification in SuSLik is given as a function signature together with a pair of pre- and post-conditions, which are both SL assertions [9]. The synthesiser generates code that satisfies the given specification, along with the SL *proof* of its correctness by searching in a space of proofs that can be derived by using the rules of the underlying logic [13]. A distinguishing feature of *Synthetic* Separation Logic is the format of its assertions. An SSL assertion consists of two parts: a *pure part* and a *spatial part*. The pure part is a Boolean expression constraining the variables of the specification using a few basic relations, like equality and the less-than relation for integers. The spatial part is a *symbolic heap*, which consists of a list of *heaplets* separated by the $*$ symbol.

Each heaplet takes on one of the following forms [9]:

- `emp`: This represents the empty heap. It is also the left and right identity for $*$.
- $\ell \mapsto a$: This asserts that memory location ℓ points to the value a. It also asserts that the location ℓ is accessible.
- $[\ell, \iota]$: At memory location ℓ, there is a block of size ι.
- $p(\overline{\phi})$: This is an application of the ***inductive predicate*** p to the arguments $\overline{\phi}$. An inductive predicate has a collection of branches, guarded by Boolean expressions (conditions) on the parameters. The body of each branch is an SSL assertion. The assertion associated with the first branch condition that is satisfied is used in place of the application. Note that inductive predicates can be (and often are) recursively defined.

The general form of an SSL assertion is $(p; h_1 * h_2 * \cdots * h_n)$, where p is the pure part and $h_1, h_2, \cdots, h_n$ are the heaplets which are the conjuncts that make up a separated conjunction

Variable	x, y Alpha-numeric identifiers $\in$ Var
Size, offset	n, ι Non-negative integers
Expression	$e ::= 0 \mid \texttt{true} \mid x \mid e = e \mid e \wedge e \mid \neg e \mid d$
$\mathcal{T}$-expr.	$d ::= n \mid x \mid d + d \mid n \cdot d \mid \{\} \mid d \mid \cdots$
Command	$c ::= \texttt{let x = *(x + } \iota \texttt{)} \mid \texttt{*(x + } \iota \texttt{) = e} \mid$
	$\texttt{let x = malloc(n)} \mid \texttt{free(x)} \mid \texttt{error} \mid \texttt{f(}\overline{e_i}\texttt{)}$
Program	$\Pi ::= \overline{f(\overline{x_i})\ \{\ c\ \}}\ ;\ c$
Logical variable	ν, ω
Cardinality variable	α
$\mathcal{T}$-term	$\kappa ::= \nu \mid e \mid \cdots$
Pure logic term	$\phi, \psi, \chi ::= \kappa \mid \phi = \phi \mid \phi \wedge \phi \mid \neg\phi$
Symbolic heap	$P, Q, R ::= \texttt{emp} \mid \langle e, \iota \rangle \mapsto e \mid [e, \iota] \mid p^{\alpha}(\overline{\phi_i}) \mid P * Q$
Heap predicate	$\mathcal{D} ::= p^{\alpha}(\overline{x_i}) : \overline{e_j \Rightarrow \exists \overline{y}.\{\chi_j; R_j\}}$
Assertion	$\mathcal{P}, \mathcal{Q} ::= \{\phi; P\}$
Environment	$\Gamma ::= \forall \overline{x_i}.\exists \overline{y_j}.$
Context	$\Sigma ::= \overline{\mathcal{D}}$
Synthesis goal	$\mathcal{G} ::= P \rightsquigarrow Q$

Figure 2 Syntax of Synthetic Separation Logic

of the spatial part. $*$ is *separating conjunction*: $h_1 * h_2$ means that the heaplets h_1 and h_2 apply to disjoint parts of the heap. A syntax definition for SSL is given in Figure 2, which is adapted from *Cyclic Program Synthesis* by Itzhaky *et al.* [6]. We will also use the symbol `**` for separating conjunction and the symbol `:->` for $\mapsto$.

As the specification language of SuSLik, SSL serves as the compilation target for the Pika language. From there, executable programs are generated through SuSLik's program synthesis. Consider a program that takes an integer x and a result in location r and stores $x + 1$ at location r. This can be written as the SuSLik specification:

```
void add1Proc(int x, loc r)
  { r :-> 0 }
  { y == x + 1 ; r :-> y }
{ ?? }
```

This example can be written as follows in our tool:

```
add1Proc : Int -> Int;
add1Proc x := x + 1;
```

In contrast with SuSLik spec, the Pika one requires no direct manipulation with pointers.

2.2 The Pika Language

While SSL provides a specification language that allows tools like SuSLik to synthesise code, it is only able to express specifications as pointers. This is useful for some applications, such as embedded systems, but it does not provide any high-level abstractions. As a result, every part of a specification is tailored to a specific memory representation of each data structure involved. To address this shortcoming, we introduce a language with algebraic data types that gets translated into SSL specifications. Additionally, we introduce a language construct that allows the programmer to specify a *memory representation* of an algebraic data type called a *layout*. The distinction between algebraic data types and layouts provides

```
data List := Nil | Cons Int List;

Sll : List >-> layout[x];
Sll (Nil) := emp;
Sll (Cons head tail) := x :-> head, (x+1) :-> tail, Sll tail;
```

Figure 3 `List` algebraic data type together with its singly-linked list layout `Sll`

the separation of concerns between the low-level representation of a data structure and code that manipulates it at a high level.

Syntactically, Pika resembles a functional programming language of the Miranda [12] and Haskell [5] lineage. It supports algebraic data types, pattern matching at top-level function definitions (though not inside expressions) and Boolean guards. The primary difference arises due to the existence of layouts and the fact that the language is compiled to an SSL specification rather than executable code. Beyond algebraic data types and layouts, Pika has a built-in type for integers as well as Booleans.

Functions in Pika are only defined by their operations on algebraic data types. Thus, all function definitions are "layout-polymorphic" over the particular choices of layouts for their arguments and result. Giving a layout polymorphic function, a particular choice of layouts is called "instantiation". Specifying the layout of a non-function value is called "lowering."

The code generator is instructed to generate a SuSLik specification for a certain function at a certain instantiation by using a `%generate` directive. For example, if there is a function definition with the type signature `mapAdd1 : List -> List`, a line reading `%generate mapAdd1 [Sll] Sll` would instruct the Pika compiler to generate the SuSLik inductive predicate corresponding to `mapAdd1` instantiated to the `Sll` layout for both its argument and its result. An example of an ADT definition and a corresponding layout definition is given in Figure 3. There is one unusual part of the syntax in particular that requires further explanation: layout type signatures. A layout definition consists of a *layout type signature* and a pattern match (much like a function definition), with lists of SSL heaplets on the right-hand sides. A layout type signature has a special form $A : \alpha \rightarrowtail \texttt{layout}[x]$. This says that the layout A is for the algebraic data type α and the SSL output variable x denoting the "head" pointer of the respective structure.

2.3 Pika by Example

We demonstrate the characteristic usages of Pika by a series of examples. In these examples, we will often make use of the `List` algebraic data type and its `Sll` layout from Figure 3. A simple example of Pika code that illustrates algebraic data types and layouts is a function which creates a singleton list out of the given integer argument:

```
%generate singleton [Int] Sll

singleton : Int -> List;
singleton x := Cons x (Nil);
```

This gets compiled to the following SuSLik specification (modulo auto-generated names):

```
predicate singleton(int p, loc r) {
| true => { r :-> p ** (r+1) :-> 0 ** [r,2] }
}
```

```
predicate sll(loc x) {
| x == 0 => { emp }
| not (x == 0) => { [x, 2] ** x :-> v ** (x+1) :-> nxt ** sll(nxt) }
}

predicate mapAdd1(loc x, loc r) {
| x == 0 => { emp }
| not (x == 0) => { [x, 2] ** x :-> v ** (x+1) :-> xNxt ** [r, 2]
    ** r :-> (v+1) ** (r+1) :-> rNxt ** mapAdd1(xNxt, rNxt) } }

void mapAdd1_fn(loc x, loc y)
  { sll(x) ** y :-> 0 }
  { y :-> r ** mapAdd1(x, r) }
{ ?? }
```

Figure 4 Specifying a function that adds one to each element of a singly-linked list in SuSLik.

A slightly more complicated example comes from trying to write a functional-style `map` function directly in SuSLik. Consider a function which adds 1 to each integer in a list of integers. Considering the list implementation to be a singly-linked list with a fixed layout, one way to express this in SuSLik is shown in Figure 4. In the `mapAdd1` predicate, the input list is given as the `x` parameter and the `r` parameter points to the output list. In the non-null case, the head of `r` is required to be the successor of the head of `x`. The predicate is then applied recursively to the tails.

Note that inductive predicates are used for *two* different purposes: the `sll` inductive predicate describes a singly-linked list data structure, while the `mapAdd1` inductive predicate describes how the input list relates to the output list. Both are used in the specification of `mapAdd1_fn`: `sll` in the precondition and `mapAdd1` in the postcondition.

Using the `mapAdd1` inductive predicate gives us two advantages over attempting to put the SSL propositions directly into the postcondition of `mapAdd1_fn`:

1. We are able to express a conditional on the shape of the list. This is much like pattern matching in a language with algebraic data types, but we are examining the pointer involved directly.
2. We are able to express *recursion* part of the postcondition: the `mapAdd1` inductive predicate refers to itself in the `not (x == 0)` branch.

These two features are both reminiscent of features common in functional programming languages: pattern matching and recursion. However, there are still some significant differences:

- In traditional pattern matching, the underlying memory representation of the data structure is not exposed.
- Compared to a functional programming language, the meaning of the specification is more obscured. It is necessary to think about the structure of the linked data structure to determine what the specification is saying. This is related to the first point: The memory representation is front-and-center.
- In many functional languages, mutation is either restricted or generally discouraged. In SuSLik, mutation is commonplace.

Suppose we want to write the functional program that corresponds to this specification. One way to do this in a Haskell-like language is by using the `List` type from Figure 3.

```
mapAdd1_fn : List -> List;
mapAdd1_fn (Nil) := 0;
mapAdd1_fn (Cons head tail) := Cons (head + 1) (mapAdd1_fn tail);
```

The only missing information is the memory representation of the `List` data structure. We do not want the `mapAdd1_fn` implementation to deal with this directly, however. We want to separate the more abstract notions of pattern matching and constructors from the concrete memory layout that the data structure has.

To accomplish this, we now extend the code with the definition of `Sll` from Figure 3. `Sll` is a *layout* for the algebraic data type `List`. Now we have all of the information of the original specification but rearranged so that the low-level memory layout is separated from the rest of the code. This separation brings us to an important observation about the language, manifested throughout these examples: none of the function definitions need to *directly* perform any pointer manipulations. This is relegated entirely to the reusable layout definitions for the ADTs. The examples are written entirely as recursive functions that pattern match on, and construct, ADTs.

All that is left is to connect these two parts: the layouts and the function definitions. We instruct a SuSLik specification generator to generate a SuSLik specification from the `mapAdd1_fn` function using the `Sll` layout:

```
%generate mapAdd1_fn [Sll] Sll
```

The `[Sll]` part of the directive tells the generator which layouts are used for the arguments. In this case, the function only has one argument and the `Sll` layout is used. The `Sll` at the end specifies the layout for the result.

2.3.1 Synthesising the in-place map function

We can generalise our `mapAdd1` to map arbitrary `Int` functions over a list and then redefine `mapAdd1` using the new `map`.

```
%generate mapAdd1 [Sll] Sll

data List := Nil | Cons Int List;

Sll : List >-> layout[x];
Sll (Nil) := emp;
Sll (Cons head tail) := x :-> head, (x+1) :-> tail, Sll tail;

map : (Int -> Int) -> List -> List;
map f (Nil) := Nil;
map f (Cons x xs) := Cons (instantiate [Int] Int f x) (map f xs);

add1 : Int -> Int;
add1 x := x + 1;

mapAdd1 : List -> List;
mapAdd1 xs := instantiate [Int -> Int, Sll] Sll map add1 xs;
```

The keyword `instantiate` gives specific layouts to use for the types in function applications, *e.g.*, if a function `g` has type `A -> B -> C -> D`, then `instantiate [L1, L2, L3] L4 g x y z` will use layout `L1` for type `A`, `L2` for type `B`, `L3` for type `C` and, finally, `L4` for the result type `D`, while applying the function `g` to the three arguments `x`, `y` and `z`.

```
predicate filterLt9(loc x, loc r) {
| (x == 0) => { r == 0 ; emp }
| not (x == 0) && head < 9 =>
    { x :-> head ** (x+1) :-> tail ** [x,2] ** filterLt9(tail, r) }
| not (x == 0) && not (head < 9) =>
    { x :-> head ** (x+1) :-> tail ** [x,2] ** filterLt9(tail, y)
      ** r :-> head ** (r+1) :-> y ** [r,2] } }

void filterLt9(loc x1, loc r)
  { Sll(x1) ** r :-> 0 }
  { filterLt9(x1, r0) ** r :-> r0 }
{ ?? }
```

Figure 5 SuSLik specification of `filterLt9`, excluding `Sll` which is given in Figure 3

This example makes use of `instantiate` in two places. In the first case where we have the call `instantiate [Int] Int f x`, the builtin `Int` layout is used for both the input and output. In this special case, the `Int` layout shares a name with the `Int` type that it represents. This is necessary since `instantiate` is used for all non-recursive calls that are not constructor applications.

In the second use, `instantiate [Int -> Int, Sll] Sll map add1 xs`, we specify that the second argument uses the `Sll` layout for the `List` type from Figure 3. We also give `Sll` as the layout for the result of the call. Note that it is not necessary to use `instantiate` for the recursive call to `map`. This is because the appropriate layout is inferred for recursive calls.

The type signature of `mapAdd1` implies that it is layout polymorphic, as the type does not refer to any specific layout. It might be surprising that `instantiate` is required in the body of `mapAdd1` since the type signature of `mapAdd1` suggests that it is layout polymorphic and yet we must pick a specific `List` layout when we use `instantiate` to call `map`. This is because, in general, a call inside the body of some function `fn` might use any layout, even layouts that have no relation to the layouts that `fn` is instantiated to. Finally, please note that our benchmarks shown in Figure 1 include a more general version of `mapAdd`.

2.3.2 Guards

While we have a pattern-matching construct at the top level of a function definition, we have not seen a way to branch on a Boolean value so far. This is a feature that is readily available at the level of SuSLik, since the same conditional construct we use to implement pattern matching can also use other Boolean expressions.

We can expose this in the functional language using a *guard*, much like Haskell's guards. Suppose we want to write a specialised filter-like function. Specifically, we want a function that filters out all elements of a list that are less than 9. This is a specific example where the SuSLik specification is noticeably more difficult to read. For a SuSLik specification of this example, see Figure 5

On the other hand, an implementation of this in Pika is:

```
%generate filterLt9 [Sll] Sll

filterLt9 : List -> List;
filterLt9 (Nil) := Nil;
filterLt9 (Cons head tail)
```

```
  | head < 9       := filterLt9 tail;
  | not (head < 9) := Cons head (filterLt9 tail);
```

When translating a guarded function body, the translator takes the conjunction of the Boolean guard condition with the condition for the pattern match. Finally, please note that our benchmarks shown in Figure 1 include a more general version of `filterLt`.

While the SuSLik version of `filterLt9` requires working with pointers directly, the Pika version uses pattern matching and constructor application. This allows the Pika code to work independent of the layout used.

2.3.3 `if-then-else`

Another feature that is common in functional languages is `if-then-else` expressions. This has a straightforward translation into SuSLik. The `if-then-else` construct corresponds to SuSLik's C-like ternary operator. We can use this feature to implement the `even` function which produces 1 is the argument is even and 0 otherwise.

```
%generate even [Int] Int

even : Int -> Int;
even (n) := if (n % 2) == 0 then 1 else 0;
```

2.3.4 Using multiple layouts

To show the interaction between multiple algebraic data types, we write a function that follows the left branches of a binary tree and collects the values stored in those nodes into a list. This example demonstrates a binary tree algebraic data type and a layout that corresponds to it.

```
%generate leftList [TreeLayout] Sll

data Tree := Leaf | Node Int Tree Tree;

TreeLayout : Tree >-> layout[x];
TreeLayout (Leaf) := emp;
TreeLayout (Node payload left right) := x :-> payload, (x+1) :-> left,
  (x+2) :-> right, TreeLayout left, TreeLayout right;

leftList : Tree -> List;
leftList (Leaf) := Nil;
leftList (Node a b c) := Cons a (leftList b);
```

2.3.5 Synthesising fold

A fold is a common kind of operation on a data structure in functional programming, where a binary function is applied to the elements of a data structure to create a summary value. For example, if the binary function is the addition function, it will give the sum of all the elements of the data structure. The classic example of such a fold is a fold on a list. In this example, we will write a right fold over a `List`.

```
%generate fold_List [Int, Sll] Int
```

```
fold_List : Int -> List -> Int;
fold_List z (Nil) := z;
fold_List z (Cons x xs) :=
  instantiate [Int, Int] Int f x (fold_List z xs);
```

We will specifically look at the specialization where we use the addition function for `f` so that we can focus on way that layouts are used in the translation. This sort of specialization corresponds to defunctionalization. The compiler produces the following SuSLik specification for `fold_List`:

```
predicate fold_List(int i1, loc x, int i2) {
  | x == 0 => { i2 == i1 ; emp }
  | not (x == 0) => { (zz4 == i1) && ((zz5 == nxt13) &&
     (i2 == (h + b3))) ; [x,2] ** x :-> h ** (x + 1) :-> nxt13 **
     fold_List(zz4, zz5, b3) } }
```

The first two parameters of the SuSLik predicate correspond to the two arguments of the Pika function. The final parameter of the predicate, `i2`, corresponds to the output of the Pika function. There are two cases:

1. The `x == 0` case corresponds to the `Nil` case in Pika. In this case, the pure part of the assertion (to the left of the semicolon) requires that the output is equal to the first parameter. This is because the Pika function returns the first parameter in its `Nil` case.
2. The `not (x == 0)` case corresponds to the `Cons` case. First, let's look at the spatial part (this is everything to the *right* of the semicolon). The pattern match destructures the `Cons` into its head and tail. Likewise, in the spatial part of the SuSLik predicate, we require that `x` points to `h` (the head) and `x + 1` points to `nxt13` (the tail). We also recursively call the predicate on the tail. The pure part does two things: it introduces new names for things (these are used internally) and it requires that the output `i2` is the sum of the head (`h`) and the value obtained from the output of the recursive call (`b3`).

3 Formal Semantics of Pika

In this section, we have the following plan:

- We define abstract machine semantics for executing a subset of Pika programs. This semantics is given by the big-step relation $\longmapsto$ which we define later.
- We define the translation from that subset of Pika into SSL. This translation is given by the function $\mathcal{T}[\![e]\!]_{V,r}$ from Pika expressions into SSL propositions. The r is a variable name to be used in the resulting proposition and V is a collection of fresh names.
- We prove a soundness theorem. Given any well-typed expression e and an abstract machine reduction producing the store-heap pair (σ', h'), the SSL translation of e should be satisfied by SSL model (σ', h'). This is stated formally, and proven, in Theorem 3.

This subset of Pika does not have guards or conditional expressions, but it does have pattern matching. It also has the requirement that functions can only have one argument. Unlike the implementation, there is no elaboration. As a result, every algebraic data type value must be lowered to a specific layout at every usage and every function application must be explicitly instantiated with a layout for the argument and a layout for the result. We also limit the available integer and Boolean operations for brevity.

The grammar for this subset is given in Figure 6. The grammar for types, layout definitions and algebraic data type definitions remain the same as before and are therefore

$\langle i \rangle ::= \cdots \mid \text{-2} \mid \text{-1} \mid 0 \mid 1 \mid 2 \mid \cdots$

$\langle b \rangle ::= \texttt{true} \mid \texttt{false}$

$\langle e \rangle ::= \langle var \rangle \mid \langle i \rangle \mid \langle b \rangle \mid \langle e \rangle + \langle e \rangle \mid \text{C } \overline{\langle e \rangle} \mid \texttt{inst}_{A,B}(f)(\langle e \rangle) \mid \texttt{lower}_A(\langle e \rangle)$

$\langle \textit{fn-def} \rangle ::= \overline{\langle \textit{fn-case} \rangle}$

$\langle \textit{fn-case} \rangle ::= \text{f } \langle \textit{pattern} \rangle := \langle e \rangle$

Figure 6 Grammar for restricted Pika subset

omitted. $\texttt{inst}_{A,B}(f)$ corresponds to `instantiate [A] B f`. We also include another construct, $\texttt{lower}_A(C\ e_1 \cdots e_n)$. This says to use the specific layout A for the given constructor application $C\ e_1 \cdots e_n$.

The semantics for SSL are largely derived [9] from standard separation logic semantics. [11]

3.1 Overview of the Two Interpretations

The soundness theorem will link the abstract machine semantics to the translation. In fact, the abstract machine semantics and the translation are similar to each other. For the abstract machine we manipulate *concrete* heaps, while for the translation we generate *symbolic* heaps.

Comparing the two further, there are two main points (beyond what we've already mentioned) where these two interpretations of Pika differ:

1. When we need to unfold a layout, how do we know which layout branch to choose?
2. How do we translate function applications (including, but not limited to, recursive applications)?

First, consider the abstract machine semantics. In this case, we are able to choose which branch of a layout to use by evaluating the expression we are applying it to until the expression is reduced to a constructor value (where a "constructor value" is either a value or a constructor applied to constructor values). If the expression is well-typed, this will always be a constructor of the algebraic data type corresponding to the layout. The two rules that this applies to are AM-Lower and AM-Instantiate. To interpret a function application, we interpret its arguments and substitute the results into the body of the function. We then proceed to interpret the substituted function body. This process is performed by the AM-Instantiate rule.

Next, consider the SSL translation. Here we can determine which layout branch to use by generating a Boolean condition that will be true if and only if the SSL proposition on the right-hand side of the branch holds for the heap. Note that we assume that the programmer-supplied layout definitions are *injective* functions from algebraic data types to SSL assertions (up to bi-implication). We can directly translate function applications into SSL inductive predicate applications. Since inductive predicates already allow for recursive applications, there is no special handling necessary for recursion.

After defining these interpretations, we show how the abstract machine relation $\longmapsto$ and the SSL interpretation function $\mathcal{T}[\![e]\!]_{V,r}$ relate to each other by the Soundness Theorem 3.

3.2 Abstract Machine Semantics

In this section, we will define an abstract machine semantics for Pika and relate this to the standard semantics for SSL.

3.2.1 Notation and Setup

The set of values is $\text{Val} = \mathbb{Z} \cup \mathbb{B} \cup \text{Loc}$. Each of these three sets is disjoint from the other two. In particular, note that Loc and $\mathbb{Z}$ are disjoint.

There is also a set of Pika values FsVal. This includes all the elements of Val, but also includes "constructor values" given by the rules in Figure 7. In addition to the store and heap of standard SSL semantics, the abstract machine semantics uses an FsStore. This is a partial function from locations to Pika values: $\text{FsStore} = \text{Loc} \rightharpoonup \text{FsVal}$. The primary purpose of this is to recover constructor values when given a location.

The general format of the transition relation is $(e, \sigma, h, \mathcal{F}) \longmapsto (v, \sigma', h', \mathcal{F}', r)$, where the expression e results in the store being updated from σ to σ', the heap being updated from h to h', v is the Val obtained by evaluating e, the initial and final FsStore are $\mathcal{F}$ and $\mathcal{F}'$ and the result is stored in variable r. We assume that there is a global environment Σ which contains all layout definition equations and function definition equations.

Given a heap h and a heap layout H, we will make use of the notation $h \cdot [H]$. This extends the heap with the location assignments given in H. We say that the layout body H is *acting* on the heap h. This is defined in Figure 9. The intuition for this is that h gets updated using the symbolic heap description in H. For example, $\emptyset \cdot [a \texttt{ :-> } 7]$ will contain only the value 7 at the location a. It is assumed that H does not have any variables on the right-hand side of `:->`.

3.2.2 Abstract Machine Rules

The abstract machine semantics provides big-step operational semantics for evaluating Pika expressions on a heap machine. Its rules, given by Figure 8, make use of standard SSL models:

Model	$\mathcal{M} ::= (\sigma, h)$
Store	$\sigma : \text{Var} \rightharpoonup \text{Val}$
Heap	$h : \text{Loc} \rightharpoonup \text{Val}$

Note that a compound expression, consisting of multiple subexpressions, uses *disjoint* parts of the heap for each subexpression. This can be seen in the AM-Add, AM-Lower and AM-Instantiate rules, which is essential for the proof of Soundness Theorem 3.

3.3 Translating Pika Specifications into SSL

We will define two translations: One from Pika *expressions* into SSL propositions and the other from Pika *definitions* into SSL inductive predicate definitions. We start with the former.

3.3.1 Translating Expressions

In the rules given in Figure 10, the notation $\mathcal{I}_{A,B}(f)$ gives the name of the inductive predicate that the Pika function f translates to when it is instantiated to the layouts A and B.

We start by defining the translation rules for expressions. We use these translation rules in Lemma 1 to define a translation function $\mathcal{T}[\![\cdot]\!]_{V,r}$. Then, we will define translation rules for function definitions. The translation relation for expressions has the form $(e, V) \Downarrow (p, s, V', v)$,

$$\frac{x \in \mathrm{Val}}{x \in \mathrm{FsVal}}\ \textsc{FsVal-Base} \qquad \frac{x_1 \in \mathrm{FsVal} \quad \cdots \quad x_n \in \mathrm{FsVal}}{(C\ x_1 \cdots x_n) \in \mathrm{FsVal}}\ \textsc{FsVal-Constr}$$

Figure 7 FsVal judgment rules

$$\frac{r \text{ fresh} \quad i \in \mathbb{Z} \quad \sigma' = \sigma \cup \{(r, i)\}}{(i, \sigma, \varnothing, \mathcal{F}) \longmapsto (i, \sigma', \varnothing, \mathcal{F}, r)}\ \textsc{AM-Int} \qquad \frac{r \text{ fresh} \quad b \in \mathbb{B} \quad \sigma' = \sigma \cup \{(r, b)\}}{(b, \sigma, \varnothing, \mathcal{F}) \longmapsto (b, \sigma', \varnothing, \mathcal{F}, r)}\ \textsc{AM-Bool}$$

$$\frac{v \in dom(\sigma) \quad \sigma(v) \notin \mathrm{Loc}}{(v, \sigma, \varnothing, \mathcal{F}) \longmapsto (\sigma(v), \sigma, \varnothing, \mathcal{F}, v)}\ \textsc{AM-Var-Base}$$

$$\frac{v \in dom(\sigma) \quad \sigma(v) \in \mathrm{Loc}}{(v, \sigma, \varnothing, \mathcal{F}) \longmapsto (\mathcal{F}(\sigma(v)), \sigma, \varnothing, \mathcal{F}, v)}\ \textsc{AM-Var-Loc}$$

$$\frac{\begin{array}{c}(x, \sigma, \mathcal{F}, h_1) \longmapsto (x', \sigma_x, h_1', \mathcal{F}, v_x) \quad (y, \sigma_x, \mathcal{F}, h_2) \longmapsto (y', \sigma_y, h_2', \mathcal{F}, v_y) \\ r \text{ fresh} \quad h = h_1 \circ h_2 \quad h' = h_1' \circ h_2' \quad z = x' + y' \quad \sigma' = \sigma_y \cup \{(r, z)\}\end{array}}{(x + y, \sigma, h, \mathcal{F}) \longmapsto (z, \sigma', h', \mathcal{F}, r)}\ \textsc{AM-Add}$$

$$\frac{\begin{array}{c}(A[x]\ (C\ a_1 \cdots a_n) := H) \in \Sigma \quad (e, \sigma_0, h_0) \longmapsto (C\ e_1 \cdots e_n, \sigma_1, h_1, y_1) \\ (e_i, \sigma_i, h_i) \longmapsto (e_i', \sigma_{i+1}, h_i', v_i) \text{ for each } 1 \le i \le n \\ h' = h_1' \circ h_2' \circ \cdots \circ h_n' \quad h = h_0 \circ h_1 \circ \cdots \circ h_n \quad \sigma' = \sigma_{n+1} \cup \{(r, \ell)\} \\ \ell \text{ fresh} \quad r \text{ fresh} \\ H' = H[x := \ell][a_1 := \sigma_2(v_1)][a_2 := \sigma_3(v_2)] \cdots [a_n := \sigma_{n+1}(v_n)] \\ \mathcal{F}' = \mathcal{F} \cup \{(\ell, (C\ e_1' \cdots e_n'))\}\end{array}}{(\mathtt{lower}_A(e), \sigma_0, h, \mathcal{F}) \longmapsto ((C\ e_1' \cdots e_n'), \sigma', h' \cdot [H'], \mathcal{F}', r)}\ \textsc{AM-Lower}$$

$$\frac{\begin{array}{c}(A[x]\ (C\ a) := H) \in \Sigma \quad (f\ (C\ b) := e_f) \in \Sigma \\ (e, \sigma, h) \longmapsto (C\ e_1, \sigma_1, h_1, y) \\ (e_1, \sigma_1, h_1) \longmapsto (e_1', \sigma_2, h_2, r) \\ \ell \text{ fresh} \quad r \text{ fresh} \quad y \text{ fresh} \\ H' = H[x := \ell][a := e_1'] \quad h' = h_1 \cdot [H'] \quad \sigma' = \sigma_f \cup \{(r, \ell)\} \\ \mathcal{F}' = \mathcal{F} \cup \{(\ell, (C\ e_1'))\} \\ (\mathtt{lower}_B(e_f[b := y]), \sigma_2, h') \longmapsto (e_f', \sigma_f, h'', r)\end{array}}{(\mathtt{inst}_{A,B}(f)(e), \sigma, \mathcal{F}, h) \longmapsto (e_f', \sigma', h'', \mathcal{F}', r)}\ \textsc{AM-Instantiate}$$

Figure 8 Abstract machine semantics rules

$$\frac{}{h \cdot [\mathtt{emp}] = h}\ \textsc{L-Emp} \qquad \frac{h' = h \cdot [H] \quad a \in \mathrm{Val}}{h \cdot [\ell\ \mathtt{:->}\ a, H] = h'[\ell \mapsto a]}\ \textsc{L-PointsTo}$$

$$\frac{e \in \mathrm{Val}}{h \cdot [A[x](e), H] = h \cdot [H]}\ \textsc{L-Apply}$$

Figure 9 Rules for layout bodies acting on heaps

where p and s are the pure part and spatial part (respectively) of an SSL assertion and $V, V' \in \mathcal{P}(\text{Var})$.

The rules can be thought of as being in two groups:

1. Rules for base type expressions, such as S-Lit and S-Add.
2. Rules for using layouts to translate expressions whose types involve algebraic data types. Examples include S-Lower-Constr and S-Inst-Inst.

In the first group, consider S-Add. In the result of the translation, we've included $v == v_1 + v_2$ in the list of conjuncts in the pure part. Here, v_1 and v_2 are the results of the two subexpressions in the addition. In the pure part, we also include the pure parts of the two subexpressions as conjuncts. These are p_1 and p_2. The spatial part of the translation consists of the spatial parts of the two subexpressions, s_1 and s_2.

Now, in the second group, consider S-Lower-Constr. This translates a Pika constructor application expression using a specific layout (which is provided by using the `lower_`$(-)$ construct). It takes the specific branch of the layout corresponding to the constructor in question and puts the right-hand side of that branch into the spatial part, after applying the appropriate substitutions for the arguments given to the constructor in the application. The right-hand side of the layout branch is H and, after the substitution, it is called H'.

The S-Inst-Inst rule is used to translate a function application being applied to the result of another function application, given particular layouts for each application. In SuSLik, it does not make sense to directly apply a predicate to another predicate application. Therefore, we must do an ANF-like translation, where the result does not have "compound" applications like this. This translation is exactly what S-Inst-Inst is doing.

▶ **Lemma 1** ($\mathcal{T}[\![\cdot]\!]$ function). *$(\cdot, V) \Downarrow (\cdot, \cdot, \cdot, r)$ is a computable function* $\text{Expr} \to (\text{Pure} \times \text{Spatial} \times \mathcal{P}(\text{Var}))$*, given fixed V and r where $r \notin V$.*

By throwing away the third element of the tuple in the codomain, we obtain a function $\text{Expr} \to (\text{Pure} \times \text{Spatial})$ *from expressions to SSL propositions.*

Call this function $\mathcal{T}[\![\cdot]\!]_{V,r}$. That is, we define the function as follows where $r \notin V$:

$$\mathcal{T}[\![e]\!]_{V,r} = (p; s) \iff (e, V) \Downarrow (p, s, V', r) \text{ for some } V'$$

We highlight the computability of this function to emphasise the fact that it can be used directly in an implementation of this subset of Pika.

3.3.2 Translating Function Definitions

The next step is to define the translation for Pika function definitions. In order to do this, we must first figure out how to determine the appropriate layout branch to use when unfolding a layout, a problem we highlighted earlier. Once this is accomplished, the rest of the translation can be defined. When this problem was solved for the abstract machine semantics, it was possible to simply evaluate the Pika expression until a constructor application expression was reached. From there, it is possible to just look at the constructor name and match it against the appropriate layout branch.

For the translation, however, we do not have the luxury of being able to evaluate expressions. Instead, we must instead rely on the fact that, in SSL, a "pure" (Boolean) condition can determine which inductive predicate branch to use. The question becomes: *Given an algebraic data type and a layout for that ADT, how do we generate an appropriate Boolean condition for a given constructor for the ADT?*

$$\frac{i \in \mathbb{Z} \quad v \text{ fresh}}{(i, V) \Downarrow (v == i, \texttt{emp}, V \cup \{v\}, v)} \text{ S-Int}$$

$$\frac{b \in \mathbb{B} \quad v \text{ fresh}}{(b, V) \Downarrow (v == b, \texttt{emp}, V \cup \{v\}, v)} \text{ S-Bool}$$

$$\frac{v \in \text{Var}}{(v, V) \Downarrow (\texttt{true}, \texttt{emp}, V, v)} \text{ S-Var}$$

$$\frac{(e_1, V_0) \Downarrow (p_1, s_2, V_1, v_1) \quad (e_2, V_1) \Downarrow (p_2, s_2, V_2, v_2) \quad v \text{ fresh}}{(e_1 + e_2, V_0) \Downarrow (v == v_1 + v_2 \wedge p_1 \wedge p_2, s_1 * s_2, V_2 \cup \{v\}, v)} \text{ S-Add}$$

$$\frac{v \in \text{Var}}{(\texttt{lower}_A(v), V) \Downarrow (\texttt{true}, A(v), V \cup \{v\}, v)} \text{ S-Lower-Var}$$

$$\frac{\begin{array}{c}(A[x]\ (C\ a_1 \cdots a_n) := H) \in \Sigma \\ (e_i, V_i) \Downarrow (p_i, s_i, V_{i+1}, v_i) \text{ for each } 1 \le i \le n \\ v \text{ fresh} \\ V' = V_{n+1} \cup \{x\} \quad H' = H[a_1 := v_1] \cdots [a_n := v_n]\end{array}}{(\texttt{lower}_A(C\ e_1 \cdots e_n), V_1) \Downarrow (p_1 \wedge \cdots \wedge p_n, H' * s_1 * \cdots * s_n, V', x)} \text{ S-Lower-Constr}$$

$$\frac{v \in V \quad r \text{ fresh}}{(\texttt{inst}_{A,B}(f)(v), V) \Downarrow (\texttt{true}, \mathcal{I}_{A,B}(f)(v, r), V \cup \{r\}, r)} \text{ S-Inst-Var}$$

$$\frac{\begin{array}{c}(A[x]\ (C\ a_1 \cdots a_n) := H) \in \Sigma \\ (e_i, V_i) \Downarrow (p_i, s_i, V_{i+1}, v_i) \text{ for each } 1 \le i \le n \\ x \text{ fresh} \\ V' = V_{n+1} \cup \{x\} \quad H' = H[a_1 := v_1] \cdots [a_n := v_n] \\ (f\ (C\ b_1 \cdots b_n) := e_f) \in \Sigma \quad e'_f = e_f[b_1 := v_1] \cdots [b_n := v_n] \\ (\texttt{lower}_B(e'_f), V_{n+1}) \Downarrow (p, s, V', r)\end{array}}{(\texttt{inst}_{A,B}(f)(C\ e_1 \cdots e_n), V_1) \Downarrow (p \wedge p_1 \wedge \cdots \wedge p_n, s * s_1 * \cdots * s_n, V', r)} \text{ S-Inst-Constr}$$

$$\frac{\begin{array}{c}(\texttt{inst}_{A,B}(g)(e), V) \Downarrow (p_1, s_1, V_1, r_1) \\ (\texttt{inst}_{B,C}(f)(r_1), V_1) \Downarrow (p_2, s_2, V_2, r_2)\end{array}}{(\texttt{inst}_{B,C}(f)(\texttt{inst}_{A,B}(g)(e)), V) \Downarrow (p_1 \wedge p_2, s_1 * s_2, V_2, r_2)} \text{ S-Inst-Inst}$$

Figure 10 Expression Translation Rules

$$\frac{\begin{array}{c}V = \{v_1, \cdots, v_n\} \text{ where } v_1, \cdots, v_n \text{ are distinct variables} \\ r \in \text{Var} \quad r \notin V \quad c = \text{cond}(A, C, x) \quad p_1 \cdots p_n \text{ fresh} \\ (p, s) = \mathcal{T}[\![\texttt{inst}_{A,B}(f)(C\ p_1 \cdots p_n)]\!]_{V,r}\end{array}}{(f\ (C\ a_1 \cdots a_n) := e) \xrightarrow{\text{fn-def}}_{A,B} (\mathcal{I}_{A,B}(f)(x, r) : c \Rightarrow \{p; s\})} \text{ FnDef}$$

Figure 11 Translation rule for function definitions

The solution is to find a Boolean condition which, given that the inductive predicate holds, is true *if and only if* the layout branch corresponding to that ADT constructor is satisfiable. In more detail, to define the branches of an inductive predicate $\mathcal{I}_A(x)$, given an ADT α, a constructor $C : \beta_1 \to \beta_2 \to \cdots \to \beta_n \to \alpha$, a layout $A : \alpha \rightarrowtail \texttt{layout}[x]$ with a branch $A[x]\ (C\ a_1 \cdots a_n) := H$ and given that $\mathcal{I}_A(x)$ holds, find a Boolean expression b with one free variable x such that $b \iff \exists \sigma, h.\ (\sigma, h) \models H$.

▶ **Lemma 2** (cond function). *There is a computable function* $\text{cond}(\cdot, \cdot)$ *that takes in any layout* $A : \alpha \rightarrowtail \texttt{layout}[x]$ *with a branch* $A[x]\ (C\ a_1 \cdots a_n) := H$ *for a given constructor* $C : \beta_1 \to \beta_2 \to \cdots \to \beta_n \to \alpha$ *and it produces a Boolean expression with one free variable* x *such that the following holds under the assumption that* $\mathcal{I}_A(x)$ *holds.*

$$\text{cond}(A, C) \iff \exists \sigma, h.\ (\sigma, h) \models H$$

Here, $\mathcal{I}_A(x)$ *is the name of the generated inductive predicate corresponding to the layout* A*.*

With this function in hand, we are now ready to define the translation for Pika function definitions. This definition is in Figure 11. In this rule, the fresh variables $p_1, \cdots, p_n$ will be substituted for $a_1, \cdots, a_n$.

3.4 Typing Rules

Typing rules for Pika expressions are given in Figure 12. These rules differ from standard typing rules for a functional language due to the existence of layouts and their associated constructors, like `instantiate` and `lower`. If an expression is well-typed, then each use of `instantiate` and `lower` only uses layouts together with the ADT that they are defined for.

The rules also make use of a *concreteness judgment*. The rules for this judgment are given in Figure 13a. The intuition of this judgment is that a type is "concrete" iff values of that type can be directly represented in the heap machine semantics. For example, an ADT type is *not* concrete because a layout has not been specified. However, once a particular layout is specified for the ADT type, it becomes concrete. Base types, like `Int`, are also concrete.

Rules for the ensuring that global definitions are well-typed are given in Figure 13b. In this figure, Δ is the set of all (global) constructor type definitions.

3.5 From Pika to SLL Specifications: Soundness of the Translation

We want to show that our abstract machine semantics and our SSL translation fit together. In particular, our abstract machine semantics should generate models that satisfy the separation logic propositions given by our SSL translation. Figure 14 gives a high-level overview of how these pieces fit together. We will give a more specific description of this in Theorem 3.

▶ **Theorem 3** (Soundness). *For any well-typed expression* e*, if* $\mathcal{T}[\![e]\!]_{V,r}$ *is satisfiable for* $V = dom(\sigma')$ *and* $(e, \sigma, h, \mathcal{F}) \longmapsto (e', \sigma', h', \mathcal{F}', r)$*, then* $(\sigma', h') \models \mathcal{T}[\![e]\!]_{V,r}$*. That is, given an expression* e *with a satisfiable SSL translation, any heap machine state that* e *transitions to (by the abstract machine semantics) will be a model for the SSL translation of* e *(cf. Figure 14).*

Proof. See the Appendices in the extended version of the paper. ◀

The fact that, at the top level, we only translate function definitions suggests an additional theorem. We want to specifically show that any possible function application is sound, in the sense just described. This immediately follows from Theorem 3.

Abbreviating $\texttt{inst}_{A,B}(f)$ as $f_{A,B}$, we can state the following theorem:

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : \texttt{Int}}\ \text{T-Int} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \texttt{Bool}}\ \text{T-Bool} \qquad \frac{(v : \alpha) \in \Gamma}{\Gamma \vdash v : \alpha}\ \text{T-Var}$$

$$\frac{(f : \alpha \to \beta) \in \Sigma}{\Gamma \vdash f : \alpha \to \beta}\ \text{T-Fn-Global}$$

$$\frac{\Gamma \vdash x : \texttt{Int} \quad \Gamma \vdash y : \texttt{Int}}{\Gamma \vdash x + y : \texttt{Int}}\ \text{T-Add}$$

$$\frac{(v : \alpha) \in \Gamma \quad (A : \alpha \rightarrowtail \texttt{layout}[x]) \in \Sigma}{\Gamma \vdash \texttt{lower}_A(v) : A}\ \text{T-Lower-Var}$$

$$\frac{\begin{array}{c}(C : \alpha_1 \to \cdots \to \alpha_n \to \beta) \in \Sigma \quad (B : \beta \rightarrowtail \texttt{layout}[x]) \in \Sigma \\ \Gamma \vdash e_i \text{ concrete}_{\alpha_i} \text{ for each } i \text{ with } 1 \le i \le n\end{array}}{\Gamma \vdash \texttt{lower}_A(C\ e_1 \cdots e_n) : B}\ \text{T-Lower-Constr}$$

$$\frac{\begin{array}{c}(A : \alpha \rightarrowtail \texttt{layout}[x]) \in \Sigma \quad (B : \beta \rightarrowtail \texttt{layout}[y]) \in \Sigma \\ \Gamma \vdash f : \alpha \to \beta \quad \Gamma \vdash e : A\end{array}}{\Gamma \vdash \texttt{inst}_{A,B}(f)(e) : B}\ \text{T-Instantiate}$$

$$\frac{(C : \alpha_1 \to \cdots \to \alpha_n \to \beta) \in \Sigma \quad \Gamma \vdash e_i : \alpha_i \text{ for each } i \text{ with } 1 \le i \le n}{\Gamma \vdash C\ e_1 \cdots e_n : \beta}\ \text{T-Constr}$$

Figure 12 Typing rules

▶ **Theorem 4** (Application soundness). *For any well-typed function application $f_{A,B}(e)$, if $\mathcal{T}[\![f_{A,B}(e)]\!]_{V,r}$ is satisfiable for $V = dom(\sigma')$ and $(f_{A,B}(e), \sigma, h, \mathcal{F}) \longmapsto (e', \sigma', h', \mathcal{F}', r)$, then $(\sigma', h') \models \mathcal{T}[\![f_{A,B}(e)]\!]_{V,r}$.*

Proof. This follows immediately from Theorem 3. ◀

4 Extensions of SuSLik

We have shown the translation from the functional specifications into SSL specifications. However, some of the SSL specifications are not supported in the original SuSLik and existing variants. In this section, we show how to extend the SuSLik to support more features to make the whole thing work. We will show the extensions on the following three aspects:

- How to describe and call an existing function within SSL predicates.

$$\frac{\Gamma \vdash e : \texttt{Int}}{e \text{ concrete}_{\texttt{Int}}}\ \text{C-Int} \qquad \frac{\Gamma \vdash e : \texttt{Bool}}{e \text{ concrete}_{\texttt{Bool}}}\ \text{C-Bool}$$

$$\frac{(A : \alpha \rightarrowtail \texttt{layout}[x]) \in \Sigma \quad \Gamma \vdash e : A}{e \text{ concrete}_{\alpha}}\ \text{C-Layout}$$

(a) Concreteness judgment rules

$$\frac{\begin{array}{c}(C : \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to \beta) \in \Delta \\ b_1 : \alpha_1, b_2 : \alpha_2, \cdots, b_n : \alpha_n \vdash e : \gamma\end{array}}{(f\ (C\ b_1 \cdots b_n) := e) \Rightarrow f : \beta \to \gamma}\ \text{G-Fn}$$

(b) Global definition typing

Figure 13 Rules for concreteness judgement and typing global definitions

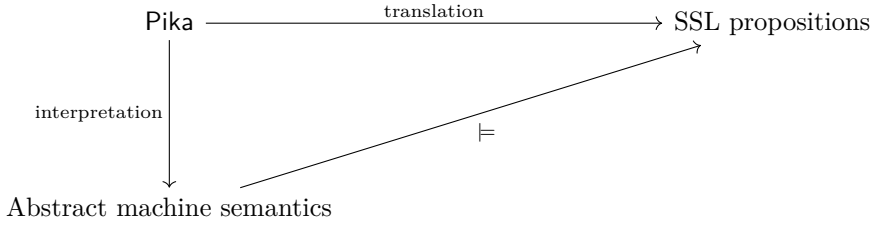


Figure 14 The relationship between the two Pika semantics given by the soundness theorem

- How to make the result of one function call as the input of another function call.
- How to synthesise programs with inductive predicates without the help of pure theory.

4.1 Function Predicates

Without any modification upon the implementation, we find the SSL predicate with some restrictions can be used to describe function relations other than data structures (named function predicates). The definition of **function predicates** is as follows:

▶ **Definition 5** (Function Predicates). Given any non-higher-order n-ary function `f(x1, ..., xn)` in the functional language, the function predicate to synthesise f has the following format:

```
predicate predf(T x1, ... ,T xn, T output){...}
```

where `T` $\in$ {`loc`, `int`}. The type of `xi` (and `output`) is decided by the type of f. If it is an integer in f, then its type is `int`; otherwise, it is `loc` (for any data structure in Pika).

Since the input of the whole workflow is functional programs, the "output" in the definition is to provide another location for the output of the function. And the specification to synthesise function f should have the following format:

```
void f(loc x1, ... ,loc xn)
{x1 :-> v1 ** x2 :-> l2 ** sll(l2) ** ... ** xn :-> vn ** output :-> 0}
{x1 :-> v1 ** x2 :-> l2 ** ... ** xn :-> vn ** output :-> output0 **
 predf(v1, l2, ..., vn, output0)}
```

4.2 SSL Rules for *func* Structure

As we show in previous examples, the reason we can have *func* structure is that the points-to structure in the post-condition is always eliminated after write operations. For example, the in-placed `inc1` functions specification is satisfied via the WRITE (Figure 15) on the location.

```
void inc_y(loc y, loc x)
{x :-> vx ** y :-> vy}
{x :-> vx + vy ** y :-> vy}
```

The core insight of *func* structure is: since the function synthesised by function predicate behaves like the pure function, it is the same as the WRITE rule in the sense that only the output location is modified. Thus, we add the new FUNCWRITE rule into the zoo of SSL rules (see Figure 15). To make the *func* structure correctly equal to some "write" operation, the following restrictions should hold, which are achieved by the translation:

- If `func f(x1, ..., xn, output)` appears in a post-condition, then no write rule can be applied to any `xi`. This is to avoid the ambiguity of the *func*.

$$\textsc{Write}\quad \frac{\mathsf{Vars}(\mathtt{e}) \subseteq \Gamma \qquad \mathtt{e} \neq \mathtt{e}' \qquad \{\phi; \mathtt{x} \mapsto \mathtt{e} * \mathtt{P}\} \rightsquigarrow \{\psi; \mathtt{x} \mapsto \mathtt{e} * \mathtt{Q}\} \,|\, \mathtt{c}}{\{\phi; \mathtt{x} \mapsto \mathtt{e}' * \mathtt{P}\} \rightsquigarrow \{\psi; \mathtt{x} \mapsto \mathtt{e} * \mathtt{Q}\} \,|\, *\mathtt{x} = \mathtt{e}\ ;\ \mathtt{c}}$$

$$\textsc{Funcwrite}\quad \frac{\forall \mathtt{i} \in [1, \mathtt{n}], \mathsf{Vars}(\mathtt{e_i}) \subseteq \Gamma \qquad \{\phi; \mathtt{P}\} \rightsquigarrow \{\psi; \mathtt{Q}\} \,|\, \mathtt{c}}{\{\phi; \mathtt{x} \mapsto \mathtt{e} * \mathtt{P}\} \rightsquigarrow \{\psi; \mathit{func}\ \mathtt{f}(\mathtt{e_1}, \ldots, \mathtt{e_n}, \mathtt{x}) * \mathtt{Q}\} \,|\, \mathtt{f}(\mathtt{e_1}, \ldots, \mathtt{e_n}, \mathtt{x})\ ;\ \mathtt{c}}$$

Figure 15 The WRITE and new FUNCWRITE rules in SSL

- The type of function f is consistent.

Note that based on the setting of the function predicate, the parameters of the function call are pointers, while the parameters of the function predicate are content to which pointers point. Furthermore, we have the *func* generated from function predicates and with the format defined in Subsection 4.1. As a result, the equivalent original SSL that duplicates points-to of one location is not a problem, since they can be merged as one.

4.3 Temporary Location for the Sequential Application

Though richly expressive, SSL has difficulty in expressing the sequential application of functions. For example, given the *func* structure available, the following function is not expressible within one function predicate:

```
f x y = g (h x) y
```

If we attempt to express it, we will have the following part in the predicate:

```
predicate f(loc x, loc y, loc output)
{... ** func h(x, houtput) ** func g(houtput, y, output)}
```

However, `houtput` is not a location in the pre-condition, which is not allowed in SSL. Thus, we introduce a new keyword `temp` to denote the temporary location for the sequential application. The new definition of `func` is as follows:

```
predicate f(loc x, loc y, loc output)
{... ** temp houtput ** func h(x, houtput) ** func g(houtput, y, output)}
```

Roughly speaking, the `temp` structure will help to allocate a new location for the output of the first function, and then use it as the input of the second function. After all appearances of `houtput` is eliminated, we will deallocate the location.

Note that the temporary variable is possible to appear in two different structures: recursive function predicates or *func* call. The reason we don't need to consider the basic arithmetic operations is that the integer will be directly used as the predicate parameter, instead of the location as the parameter. For example, the sum of a list can be expressed as:

```
predicate sum(loc l, int output){
| l == 0 => {output == 0; emp}
| l != 0 => {output == output1 + v; [l, 2] ** l :-> v ** l + 1 :-> lnxt **
    sum(lnxt, output1)} }
```

Such sequential application is common in functional programming, especially in the recursive function. For example, it is not elegant to flatten a list of lists without the sequential application.

$$\textsc{TempFunCalloc}\quad \frac{\{\phi; \mathtt{x} \mapsto \mathtt{a} * \mathtt{P}\} \rightsquigarrow \{\psi; \mathit{func}\ f(e_1, \ldots, e_n, x) * \mathit{temp}(x, 1) * Q\} \mid \mathtt{c}}{\{\phi; \mathtt{P}\} \rightsquigarrow \{\psi; \mathit{func}\ \mathtt{f}(\mathtt{e_1}, \ldots, \mathtt{e_n}, \mathtt{x}) * \mathtt{temp}(\mathtt{x}, 0) * \mathtt{Q}\} \mid \mathtt{let\ x\ =\ malloc(1)\ ;\ c}}$$

$$\textsc{TempFunCfree}\quad \frac{\{\phi; \mathtt{P}\} \rightsquigarrow \{\psi; Q\} \mid \mathtt{c}}{\nexists \mathtt{x} \in \mathtt{Q} \ \wedge\ \{\phi; \mathtt{P}\} \rightsquigarrow \{\psi; \mathtt{temp(x, 1)} * \mathtt{Q}\} \mid \quad \mathtt{let\ x0\ =\ *x\ ;\ type\_free(x0);\ free(x);\ c}}$$

Figure 16 New allocating and deallocating rule for `temp` in SSL

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

We can express this function, but with some strange structure to store all temporary lists.

```
predicate flatten(loc x, loc output){
| x == 0 => {output :-> 0}
| x != 0 => {[x, 2] ** x :-> x0 ** sll(x0) ** x + 1 :-> xnxt **
[output, 2] ** func append(x, outputnxt, output) ** output + 1 :-> outputnxt **
flatten(xnxt, outputnxt)} }
```

With such a function predicate, though we can synthesise the function whose result stored in `output` is the flattened list, the list `output` is containing a lot of intermediate values, which is neither consistent with the definition in the source language nor space efficient.

The new rules consist of allocating and deallocating rules (Figure 16). Based on the definition of the *func* structure and the function predicate, the allocated locations are different, where the `temp` location for *func* is directly used; while the `temp` location for function predicate should allocate a new location for function predicates. As for the deallocation, not only the `temp` location(s) but also the content they point to should be deallocated. That is the reason we have the `type_free` function, which is syntax sugar to deallocate the content of a location based on the type information. For example, if the type of the location is `tree`, then the `type_free` will deallocate the content of the location via `tree_free` function, which is synthesised based on the SSL predicate `tree` as follows.

```
void tree_free(loc x)
  {tree(x)}
  {emp}
```

Specifically, if the location contains the value with type `int`, then the `type_free` will do nothing. Thus, the function predicate with `temp` is much better, in the sense that no extra space is used, and the synthesised function is consistent with the source language.

```
predicate flatten(loc x, loc output){
| x == 0 => {output :-> 0}
| x != 0 => {[x, 2] ** x :-> x0 ** sll(x0) ** x + 1 :-> xnxt ** temp outputnxt
  ** flatten(xnxt, outputnxt) ** func append(x, outputnxt, output)} }
```

4.4 Avoiding Excessive Heap Manipulation with Read-Only Locations

The existing SuSLik depends on the set theory to express the pure relation. However, it is not trivial to automatically generate the pure part of SSL specifications from the functional specifications. To see why the set theory is needed, the following simple example shows the functionality of the set theory, with `sll_n` being the **s**ingle-**l**inked **l**ist with **n**o set.

```
predicate sll_n(loc x) {
|  x == 0        => {true; emp }
|  not (x == 0)  => { [x, 2] ** x :-> v ** (x + 1) :-> xnxt ** sll(xnxt) } }
predicate copy(loc x, loc y) {
|  x == 0        => {y == 0; emp }
|  not (x == 0)  => { [y, 2] ** y :-> v ** (y + 1) :-> ynxt ** [x, 2] ** x :-> v
     ** (x + 1) :-> xnxt ** copy(xnxt, ynxt) } }
```

While the intent of the function predicate `copy` is to copy the list `x` to `y`, without the set theory, the output program will be somewhat surprising to see:

```
{sll_n(x)}
void copy (loc x, loc y) {
  if (x == 0) {
  } else {
    let n = *(x + 1); copy(n, y); let y01 = *y; let y0 = malloc(2); *y = y0;
    *(y0 + 1) = y01; let vy = *y0 *x = vy; } }
{copy(x, y)}
```

The problem here is that, when we have the pure relation in the predicate to indicate that the values are the same, the synthesiser finds another possible way: instead of copying the value of `x` to `y`, we can just change the value of x to initial value `vy` after `malloc`. This is not the user intent, and the output program is not correct. Turns out, the solution is not that difficult: we simply need add a new kind of heaplet in the specification language, call *constant points-to*, which has a similar idea as read-only borrows [2]. The only difference of the constant points-to from the original *points-to* heaplet is that the value of the location is constant, which means that the WRITE rule in SSL is not applicable. By this way, the extended SuSLik will not consider the modification of the input location, thus provides the correctness mechanism (in Subsection 3.5) for the translation of Pika.

5 Evaluation

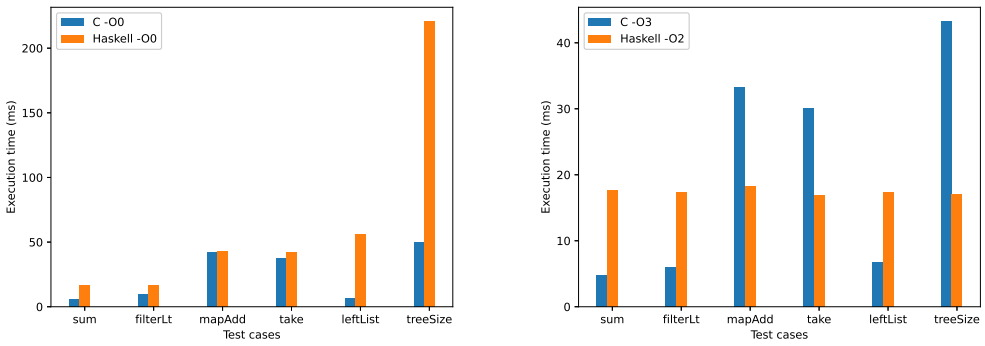
In this section, we evaluate Pika's expressiveness. A secondary objective is to evaluate Pika's performance. The performance evaluation is done largely to put Pika into context by comparing it to a prominent functional programming language (Haskell). The main purpose of Pika is to increase the expressiveness of SuSLik, which is the reason for the primary evaluation objective. Towards these goals, we answer the following research questions:

- **RQ1**: Is the performance of the synthesised code competitive with code generated from traditional functional language compilers?
- **RQ2**: In concrete terms, how does the tool's expressivity compare with the expressivity of SuSLik specifications for programs written in a functional style?
- **RQ3**: What are the failure modes of our approach?

Our implementation and benchmarks are available in supplementary material. The experiments were conducted on a 2021 MacBook Pro with an M1 processor and 32 GB of RAM. We used GHC version 9.8.1 and Apple Clang version 13.0.0.

For **RQ1**, we run benchmarks and compare execution time. The benchmarks we selected are a series of functions that manipulate data structures such as lists and trees, which covers different common abstract operations (like `map`, `filter`, `fold`). The comparison is between Haskell functions based on user-defined data structures and C functions generated from Pika's specifications (via the extended SuSLik), parameterised with the data structures of large size. We recorded both execution time with and without optimisation of compilers.

The results are shown in Figure 17. Our findings are as follows:



(a) Benchmarks without optimizations **(b)** Benchmarks with optimizations

Figure 17 Performance of C functions generated by Pika compared to Haskell/GHC

- When compiled to an executable run without optimisations, the C programs generated by Pika are faster than Haskell programs. And we can also observe that the speed difference is larger for functions with more complex data structures.
- When compiled to an executable with optimisations:
  - For functions with complex data structures, the comparison is similar to the case without optimisation.
  - For functions for the singly-linked list GHC's optimisation is very powerful, resulting in much better performance than C programs generated by Pika. With more tests, we found out the performance after GHC's optimisation is similar to the one using Haskell's built-in list. We believe this is because GHC's optimisations are fine-tuned to optimise code manipulating list-like data structures and use the same optimisation as for Haskell's built-in list. That said, similar observations regarding GHC do not hold on other complex data structures, such as trees.

To answer **RQ2**, we first find some common patterns for Pika programs in the benchmarks shown in Table 1: (1) pattern matching on ADTs (#9, 10, etc.), (2) code reusability (#3 vs 5, #7 vs 8). Those features are not directly expressible in SuSLik because of the low-level nature of SSL. For example, the `add1Head` and `add1HeadDLL` functions share the same function definition, where the only difference is the type layout used; but the SuSLik specifications need to be treated separately, which makes the codes more complex. To make some objective observations on the expressivity, we measure the number of nodes in the input Pika AST and find it consistently fewer than the number of AST nodes in the generated SuSLik specification.

To address **RQ3**, note that a particular failure mode occurs when Pika source code reuses a variable in a way that violates SSL constraints. For example, see the `selfAppend` example in Figure 18 which uses its argument twice. We could have addressed this issue by introducing a lightweight linear type system to Pika, but have not carried out this exercise yet. Another kind of failure occurs when SuSLik fails to synthesise an implementation of the generated specification: handling these failures is beyond the scope of this work.

Table 1 Statistics on the benchmarks: Pika spec size v, generated SSL spec size, translator performance, and synthesiser performance. All times are in seconds.

#	Task Name	Pika AST	SuSLik AST	$\frac{\lvert\text{Pika AST}\rvert}{\lvert\text{SuSLik AST}\rvert}$	Compile Time	Synthesis Time
1	`cons`	76	123	0.618	0.012	5.134
2	`plus`	46	91	0.505	0.002	5.992
3	`add1Head`	64	109	0.587	0.006	5.114
4	`listId`	62	107	0.579	0.005	4.906
5	`add1HeadDLL`	74	146	0.507	0.007	10.618
6	`even`	19	42	0.452	0.001	3.999
7	`foldr`	63	113	0.558	0.005	5.454
8	`sum`	58	103	0.563	0.004	5.125
9	`filterLt`	84	147	0.571	0.009	6.104
10	`mapAdd`	71	116	0.612	0.007	5.031
11	`leftList`	116	156	0.744	0.008	7.722
12	`treeSize`	78	126	0.619	0.007	5.980
13	`take`	119	212	0.561	0.012	10.447

```
selfAppend : List -> List;
selfAppend xs := instantiate [Sll, Sll] Sll append xs xs;

append : List -> List -> List;
append (Nil) ys := ys;
append (Cons x xs) ys := instantiate [Int, Sll[mutable]] Sll cons
                                       (addr x) (append xs ys);
```

Figure 18 A Pika specification not supported by SuSLik.

6 Discussion

We have given a translation from a high-level functional language into SSL specifications to be given to a program synthesiser.

In doing so, we have revealed a close connection between, on one hand, algebraic data types and recursive pattern-matching functions and, on the other hand, SSL inductive predicates. The soundness of this connection is demonstrated by Theorem 3 in Section 3.

Beyond the theory, this connection can be exploited in three directions:

- Increased type safety: Algebraic data types allow you to distinguish between types that have the same runtime heap representation.
- More reusability: An algebraic data type can have multiple layouts, each of which gives a different possible runtime heap representation for the ADT. As Pika functions are defined only in terms of algebraic data types, they naturally get the *polymorphism* of the ADT by being able to work with any layout of the ADT.
- Greater succinctness: When working at this higher level of abstraction, it generally takes less code as you are not frequently manipulating heap locations. The only mention of heap locations is in reusable layout definitions. This can also give greater clarity.

7 Related Work

Pika is built upon the SuSLik synthesis framework. SuSLik provides a synthesis mechanism for heap-manipulating programs using a variant of Separation Logic [9]. However, it does not have any high-level abstractions. In particular, writing SuSLik specifications directly involves a significant amount of pointer manipulation. Further, it does not provide abstraction over specific memory layouts. As described in Subsection 2.2, Pika addresses these limitations.

Dargent language [1] also includes a notion of layouts and layout polymorphism for a class of algebraic data types, which differs from our treatment of layouts in two primary ways:

1. In Pika, abstract memory locations (with offsets) are used. In contrast, Dargent uses offsets that are all relative to a single "beginning" memory location. The Pika approach is more amenable to heap allocation, though this requires a separate memory manager of some kind. This is exposed in the generated language with `malloc` and `free`. On the other hand, the technique taken by Dargent allows for greater control over memory management. This makes dealing with memory more complex for the programmer, but it is no longer necessary to have a separate memory manager.
2. Algebraic data types in the present language include *recursive* types and, as a result, Pika has recursive layouts for these ADTs. This feature is not currently available in Dargent.

Furthermore, layout polymorphism also works differently. While Dargent tracks layout instantiations at a type-level with type variables, in the present work we simply only check to see if a layout is valid for a given type when type-checking. In particular, we cannot write type signatures that *require* the same layout in multiple parts of the type (for instance, in a function type `List -> List` we have no way at the type-level of requiring that the argument `List` layout and the result `List` layout are the same). This more rudimentary approach that Pika currently takes could be extended in future work. Overall, the examples in the Dargent paper tend to focus on the manipulation of integer values. In contrast, we have focused largely on data structure manipulation, which follow the primary motivation of SuSLik.

Synquid is another synthesis framework with a functional surface language. While Synquid allows an even higher-level program specification than Pika through its liquid types, it does not provide any access to low-level data structure representation. [8] In contrast, Pika's level of abstraction is similar to that of a traditional functional language but, similar to Dargent, it also allows control over the data structure representation in memory.

8 Future Work and Conclusion

We make the following observations based on our experience of developing and using Pika.

By allowing layouts to use multiple SSL parameters, we would be able to give a greater variety of layouts associated with an ADT. For instance, the `List` data type used in the examples could have a doubly-linked list layout in addition to the singly-linked list layout `Sll`. Note that any existing Pika function defined over `List` will continue to work with no modification with these new layouts. Defunctionalisation and lambda lifting can also be used to implement true higher-order functions.

It is possible to do inference of some layouts, for example in `mapAdd1` we would usually want to use the same layout as the argument layout, but we leave this for future work. Another approach is to introduce type variables that correspond to layouts, as done in the series of works on the Dargent tool [1]. We leave this approach for future work as well.

Reverse transformation deserves further investigation: if we go from an SSL specification to Pika program and then compile to, *e.g.*, C, can we synthesise additional programs that traditional SSL synthesisers would struggle with? What are the limitations of this approach?

We may be able to expose more of the synthesis mechanism in the Pika language. For example, generate an SSL specification given only a Pika type signature (and corresponding `%generate` directive). This could combine well with additional polymorphism, as we could utilise the free theorems that are given by a polymorphic type signature to further constrain the resulting specification.

Finally, is it possible to derive translations for languages such as Pika from abstract machine semantics? In this paper, we have given a language with abstract machine semantics. We then give a translation of that language into SSL. We then show that the final states given by the abstract machine semantics are models for the SSL propositions produced by our translation. But is it possible to begin by specifying the abstract machine semantics and then mathematically (or automatically) *derive* an appropriate translation into SSL, with the requirement that the translation satisfies the soundness theorem?

In conclusion, we have presented Pika: a high-level functional specification language that paves the way for the efficient synthesis of a verifiably correct imperative code with in-place memory updates that is comparable in efficiency to the handwritten C.

References

1 Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Dargent: A silver bullet for verified data layout refinement. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571240.

2 Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP*, volume 12075 of *LNCS*, pages 141–168. Springer, 2020. doi:10.1007/978-3-030-44914-8_6.

3 Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. The Glasgow Haskell Compiler: A Retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 62–71. Springer, 1992. doi:10.1007/978-1-4471-3215-8_6.

4 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

5 Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, page 12–1–12–55. ACM, 2007. doi:10.1145/1238844.1238856.

6 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *PLDI*, page 944–959. ACV, 2021. doi:10.1145/3453483.3454087.

7 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

8 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, page 522–538. ACM, 2016. doi:10.1145/2908080.2908093.

9 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi:10.1145/3290385.

10 John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

11 Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*, page 53–65. ACM, 2017. doi:10.1145/3018610.3018623.

12 David Turner. An Overview of Miranda. *SIGPLAN Not.*, 21(12):158–166, 1986. doi:10.1145/15042.15053.

13 Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473589.