A New Functional-Logic Compiler for Curry: SPRITE ⋆

Sergio Antoy and Andy Jost

Computer Science Dept., Portland State University, Oregon, U.S.A.

antoy@cs.pdx.edu
ajost@pdx.edu

Abstract. We introduce a new native code compiler for Curry codenamed SPRITE. SPRITE is based on the Fair Scheme, a compilation strategy that provides instructions for transforming declarative, non-deterministic programs of a certain class into imperative, deterministic code. We outline salient features of SPRITE, discuss its implementation of Curry programs, and present benchmarking results. SPRITE is the first-to-date operationally complete implementation of Curry. Preliminary results show that ensuring this property does not incur a significant penalty.

Keywords: Functional logic programming, Compiler implementation, Operational completeness

1 Introduction

The functional-logic language Curry [16, 18] is a syntactically small extension of the popular functional language Haskell. Its seamless combination of functional and logic programming concepts gives rise to hybrid features that encourage expressive, abstract, and declarative programs [5, 18].

One example of such a feature is a functional pattern [3], in which functions are invoked in the left-hand sides of rules. This is an intuitive way to construct patterns with syntactically-sugared high-level features that puts patterns on a more even footing with expressions. In Curry, patterns can be composed and refactored like other code, and encapsulation can be used to hide details. We illustrate this with function `get`, defined below, which finds the values associated with a key in a list of key-value pairs.

```
with x = _ ++ [x] ++ _
get key (with (key, value)) = value
```
(1)

Operation `with` generates all lists containing `x`. The anonymous variables, indicated by "`_`", are place holders for expressions that are not used. Function "++" is the list-appending operator. When used in a left-hand side, as in the rule for `get`, operation `with` produces a pattern that matches any list containing `x`. Thus, the second argument to `get` is a list — any list — containing the pair `(key, value)`. The repeated variable,

⋆ This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249.

`key`, implies a constraint that, in this case, ensures that only values associated with the given key are selected.

By similar means, we may identify keys:

```
key_of (with (key, _)) = key
```
(2)

This non-deterministically returns a key of the given list; for example:

```
> key_of [('a',0), ('b',1), ('c',2)]
'a'
'b'
'c'
```
(3)

This is just one of many features [5, 18] that make Curry an appealing choice, particularly when the desired properties of a program result are easy to describe, but a set of step-by-step instructions to obtain the result is more difficult to come by.

This paper describes work towards a new Curry compiler we call Sprite. Sprite aims to be the first operationally complete Curry compiler, meaning it should produce all values of a source program (within time and space constraints). Our compiler is based on a compilation strategy named the Fair Scheme [7] that sets out rules for compiling a functional-logic program (in the form of a graph rewriting system) into abstract *deterministic* procedures that easily map to the instructions of a low-level programming language. Section 2 introduces Sprite at a high level, and describes the transformations it performs. Section 3 describes the implementation of Curry programs as imperative code. Section 4 contains benchmark results. Section 5 describes other Curry compilers. Section 6 addresses future work, and Section 7 contains our concluding remarks.

2 The Sprite Curry Compiler

Sprite is a native code compiler for Curry. Like all compilers, Sprite subjects source programs to a series of transformations. To begin, an external program is used to convert Curry source code into a desugared representation called FlatCurry [17], which Sprite further transforms into a custom intermediate representation we call ICurry. Then, following the steps laid out in the Fair Scheme, Sprite converts ICurry into a graph rewriting system that implements the program. This system is realized in a low-level, machine-independent language provided by the open-source compiler infrastructure library LLVM [22]. That code is then optimized and lowered to native assembly, ultimately producing an executable program. Sprite provides a convenience program, `scc`, to coordinate the whole procedure.

2.1 ICurry

ICurry, where the "I" stands for "imperative," is a form of Curry programs suitable for translation into imperative code. ICurry is inspired by FlatCurry [17], a popular representation of Curry programs that has been very successful for a variety of tasks including implementations in Prolog [19]. FlatCurry provides expressions that resemble

those of a functional program — e.g., they may include local declarations in the form of let blocks and conditionals in the form of case constructs, all possibly nested. Although the pattern-matching strategy is made explicit through case expressions, FlatCurry is declarative. ICurry's purpose is to represent the program in a more convenient imperative form — more convenient since Sprite will ultimately implement it in an imperative language. In imperative languages, local declarations and conditionals take the form of statements while expressions are limited to constants and/or calls to subroutines, possibly nested. ICurry provides statements for local declarations and conditionals. It provides expressions that avoid constructs that cannot be directly translated into the expressions of an imperative language.

In ICurry all non-determinism — including the implicit non-determinism in high-level features, such as functional patterns — is expressed through choices. A choice is the archetypal non-deterministic function, indicated by the symbol "?" and defined by the following rules:

```
x ? _ = x
_ ? y = y
```
(4)

The use of only choices is made possible, in part, by a duality between choices and free variables [4, 23]: any language feature expressed with choices can be implemented with free variables and vice versa. Algorithms exists to convert one to the other, meaning we are free to choose the most convenient representation in Sprite.

Finally, as in FlatCurry, the pattern-matching strategy in ICurry is made explicit and guided by a definitional tree [1], a structure made up of stepwise case distinctions that combines all rules of a function. We illustrate this for the `zip` function, defined as:

```
zip [] _ = []
zip (_:_) [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```
(5)

The corresponding definitional tree is shown below as it might appear in ICurry.

```
zip = \a b -> case a of
    []     -> []
    (x:xs) -> case b of
        []     -> []
        (y:ys) -> (x,y) : zip xs ys
```
(6)

2.2 Evaluating ICurry

It is understood how to evaluate the right-hand side of (6) efficiently; the Spineless Tagless G-machine (STG) [28], for instance, is up to the task. But the non-deterministic properties of functional-logic programs complicate matters. To evaluate `zip`, its first argument must be reduced to head-normal form. In a purely functional language, the root node of a head-normal form is always a data constructor symbol (assuming partial application is implemented by a data-like object), or else the computation fails. But for

functional-logic programs, two additional possibilities must be considered, leading to an extended case distinction:

```
zip = \a b -> case a of
   x ? y  -> (pull-tab)     - - implied
   ⊥      -> ⊥              - - implied
   []     -> []
   (x:xs) -> case b of ...
```
(7)

The infrastructure for executing this kind of pattern matching very efficiently by means of dispatch tables will be described shortly, but for now we note two things. First, there is no need for ICurry to spell out these extra cases, as they can be generated by the compiler. Second, their presence calls for an expanded notion of the computation that allows for additional node states. Because of this, Sprite hosts computations in a graph whose nodes are taken from four classes: *constructors*, *functions*, *choices*, and *failures*. Constructors and functions are provided by the source program; choices are built-in; and failures, denoted "⊥", arise from incompletely defined operations such as `head`, the function that returns the head of a list. For example, `head []` rewrites to "⊥". A simple replacement therefore propagates failure from needed arguments to roots.

Choices execute a special step called a *pull-tab* [2, 9]. Pull-tab steps lift non-determinism out of needed positions, where they prevent completion of pattern matches. The result is a choice between two more-definite expressions. A pull-tab step is shown below:

```
zip (a ? b) c → zip a x ? zip b x where x = c
```
(8)

A pattern match cannot proceed while `(a ? b)` is the first argument to `zip` because there is no matching rule in the function definition (one cannot exist because the choice symbol is disallowed on left-hand sides). We do not want to choose between `a` and `b` because such a choice would have to be reconsidered to avoid losing potential results. The pull-tab transformation "pulls" the choice to an outermore position, producing two new subexpressions, `zip a c` and `zip b c`, that can be evaluated further. The fact that `c` is shared in the result illustrates a desirable property: that node duplication is minimal and localized. Pull-tabbing involves some technicalities that we address later. The complete details are in [2].

Due to the extra cases, additional node types, and, especially, the unusual mechanics of pull-tabbing steps, we chose to develop in Sprite a new evaluation machine from scratch rather than augment an existing one such as STG. The property of pull-tabbing that it "breaks-out" of recursively-descending evaluation into nested expressions fundamentally changes the computation so that existing functional strategies are difficult to apply. In Sprite, we have implemented *de novo* an evaluation mechanism and runtime system based on the Fair Scheme. These are the topic of the next section.

3 Implementation

In this section, we describe the implementation of Curry programs in imperative code. Sprite generates LLVM code, but we assume most readers are not familiar with that. So, rather than presenting the generated code, we describe the implemented programs in terms of familiar concepts that appear directly in LLVM. In this way, the reader can think in terms of an unspecified target language — one similar to assembly — that implements those concepts. To facilitate the following description, we indicate in parentheses where a similar feature exists in the C programming language.

In the target language, values are strongly typed, and the types include integers, pointers, arrays, structures and functions. Programs are arranged into compilation units called modules that contain symbols. Symbols are visible to other modules, and to control access to them each one is marked internal (`static`) or external (`extern`). Control flow within functions is carried out by branch instructions. These include unconditional branches (`goto`), conditional branches (`if`, `for`, `while`) and indirect branches (`goto*`). The target of every branch instruction is a function-local address (label). A call stack is provided, and it is manipulated by call and return instructions that enter and exit functions, respectively. Calls are normally executed in a fresh stack frame, but the target language also supports explicit tail recursion, and Sprite puts it to good use.

3.1 Expression Representation

The expressions evaluated by a program are graphs consisting of labeled nodes having zero or more successors. Each node belongs to one of four classes, as discussed in the previous section. For constructors and functions, node labels are equivalent to symbols defined in the source program. Failures and choices are labeled with reserved symbols. Successors are references to other nodes. The number of successors, which equals the arity of the corresponding symbol, is fixed at compile time. Partial applications are written in *eval/apply* form [26].

Sprite implements graph nodes as heap objects. The layout of a heap object is shown in Fig. 1. The label is implemented as a pointer to a static info table that will be described later. Sprite emits exactly one table for each symbol in the Curry program. Successors are implemented as pointers to other heap objects.

3.2 Evaluation

Evaluation in Sprite is the repeated execution of rewriting and pull-tabbing steps. Both are implemented by two interleaved activities: replacement and pattern-matching. A replacement produces a new graph from a previous one by replacing a subexpression matching the left-hand side of a rule with the corresponding right-hand side. For instance, 1 + 1 might be replaced with 2. A replacement is implemented by overwriting the heap object at the root of the subexpression being replaced. The key advantage of this destructive update is that no pointer redirection [12, Def. 8][15] is required during a rewrite step. Reusing a heap object also has the advantage of saving one memory allocation and deallocation per replacement, but requires that every heap object be capable of storing any node, whatever its arity. Sprite meets this requirement by providing in heap

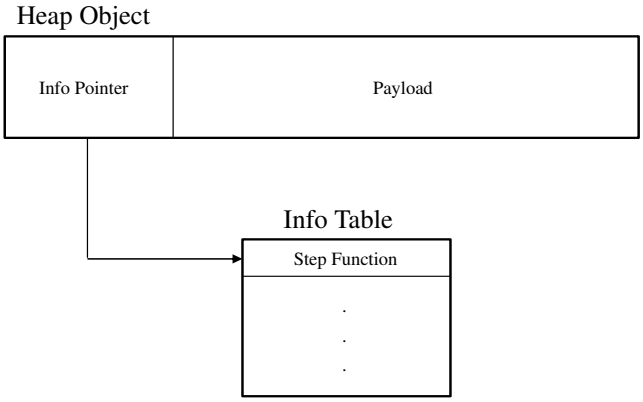

Fig. 1: The heap object layout.

objects a fixed amount of space capable of holding a small number of successors. For nodes with more successors than would fit in this space, the payload instead contains a pointer to a larger array. This approach simplifies memory management for heap objects: since they are all the same size, a single memory pool suffices. Because arities are known at compile time, no runtime checks are needed to determine whether successor pointers reside in the heap object.

Pattern-matching consists of cascading case distinctions over the root symbol of the expression being matched that culminate either in a replacement or in the patter match of a subexpression. The Fair Scheme implements this according to a strategy guided by the definitional trees encoded in ICurry. Case distinction as exemplified in (7) assumes that an expression being matched is not rooted by a function symbol. Thus, when a node needed to complete a match is labeled by a function symbol, the expression rooted by that node is evaluated until it is labeled by a non-function symbol. A function-labeled node, *n*, is evaluated by a target function called the *step* function that performs a pattern match and replacement at *n*. Each Curry function gives rise to one target function, a pointer to which is stored in the associated info table (see Fig. 1).

Operationally, pattern-matching amounts to evaluating nested case expressions similar to the one shown in (7). Sprite implements this through a mechanism we call *tagged dispatch*. With this approach, the compiler assigns each symbol a tag at compile time. Tags are sequential integers indicating which of the four classes discussed earlier the node belongs to. The three lowest tags are reserved for functions, choices, and failures (all functions have the same tag). For constructors, the tag additionally indicates *which* constructor of its type the symbol represents. To see how this works, consider the following type definition:

```
data ABC = A | B | C                                    (9)
```

`ABC` comprises three constructors in a well-defined order (any fixed order would do). To distinguish between them, SPRITE tags these with sequential numbers starting at the integer that follows the reserved tags. So, the tag of `A` is one less than the tag of `B`, which is one less than the tag of `C`. These values are unique within the type, but not throughout the program: the first constructor of each type, for instance, always has the same tag. Following these rules, it is easy to see that every case discriminator is a node tagged with one of $3+N$ consecutive integers, where N is the number of constructors in its type. To compile a case expression, SPRITE emits a jump table that transfers control to a code block appropriate for handling the discriminator tag. For example, the block that handles failure rewrites to failure, and the block that handles choices executes a pull-tab. This is shown schematically in Fig. 2. It is in general impossible to know at compile time which constructors may be encountered when the program runs, so the jump table must be complete. If a functional logic program does not define a branch for some constructor — i.e., a function is not completely defined — the branch for that constructor is a rewrite to failure.

To implement tagged dispatch, SPRITE creates function-local code blocks as labels, constructs a static jump table containing their addresses, and executes indirect branch instructions — based on the discriminator tag — through the table. Figure 3 shows a fragment of C code that approximates this. Case distinction occurs over a variable of `List` type with two constructors, `nil` and `cons`. Five labeled code blocks handle the five tags that may appear at the case discriminator. A static array of label address implements the jump table. This example assumes the *function*, *choice*, *failure*, *nil*, and *cons* tags take the values zero through four, respectively. The jump table contains one extra case not depicted in (7). When the discriminator is a *function*, the step function of the discriminator root label is applied as many times as necessary until the discriminator class is no longer *function*.

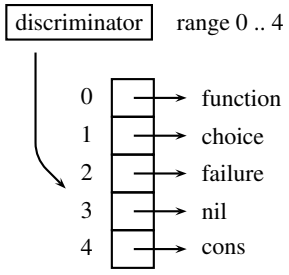

Fig. 2: Schematic representation of the SPRITE tagged dispatching mechanism for a distinction of a `List` type.

3.3 Completeness and Consistency

SPRITE aims to be the first complete Curry compiler. Informally, complete means the program produces all the intended results of the source program. More precisely, and especially for infinite computations, an arbitrary value will eventually be produced, given enough resources. This is a difficult problem because a non-terminating computation for obtaining one result could block progress of some other computation that would obtain another result. For example, the following program has a result, 1, that can be obtained in only a couple of steps, but existing Curry compilers fail to produce it:

```
loop = loop
main = loop ? (1 ? loop)
```
(10)

The Fair Scheme defines a complete evaluation strategy. It creates a work queue containing all of the expressions that might produce a result. At all times, the expression at the head of the queue is active, meaning it is being evaluated. Initially, the work queue contains only the goal expression. Whenever pull-tabbing places a choice at the root of an expression, that expression forks. It is removed from the queue, and its two alternatives are added. Whenever an expression produces a value, it is removed from the queue. To avoid endlessly working on an infinite computation, the program rotates the active computation to the end of the work queue every so often. In so doing, SPRITE guarantees that no expression is ignored forever, hence no potential result is lost.

A proof of correctness of compiled programs is provided in [7] for the abstract formulation of the compiler, the Fair Scheme. In this domain, correctness is the property that an executable program produces all and only the values intended by the corresponding source program. A delicate point is raised by pull-tabbing. A pull-tab step may duplicate or clone a choice, as the following example shows. Cloned choices should be

```
static void* jump_table[5] = {
    &function_tag, &choice_tag, &failure_tag, &nil_tag, &cons_tag
  };

entry:          goto* jump_table[discriminator.tag];
function_tag:   call_step_function(discriminator);
                goto* jump_table[discriminator.tag];
choice_tag:     /*execute a pull tab*/
failure_tag:    /*rewrite to failure*/
nil_tag:        /*rewrite to []*/
cons_tag:       /*process the nested case expression*/
```

Fig. 3: An illustrative implementation in C of the case expression shown in (7). This code fragment would appear in the body of the step function for `zip`. Variable `discriminator` refers to the case discriminator. Label `entry` indicates the entry point into this case expression.

seen as a single choice. Thus when a computation reduces a choice to its right alternative, it should also reduce any other clone of the same choice to the right alternative, and likewise for the left alternative. Computations obeying this condition are called *consistent*.

```
xor x x where x = T ? F
    →pull−tab (xor T x) ? (xor F x) where x = T ? F          (11)
```

In the example above, a pull-tab step applied to the choice in `x` leads to its duplication. Now, when evaluating either alternative of the topmost choice, a consistent strategy must recognize that the remaining choice (in `x`) is already made. For instance, when evaluating `xor T x`, the value of `x` can only be `T`, the left alternative, because the left alternative of `x` has already been selected to obtain `xor T x`. To keep track of clones, the Fair Scheme annotates choices with identifiers. Two choice nodes with identical identifiers represent the same choice. Fresh identifiers are assigned when new choices arise from a replacement; pull-tab steps copy existing identifiers. Every expression in the work queue owns a fingerprint, which is a mapping from choice identifiers to values in the set {*left,right,either*}. The fingerprint is used to detect and remove inconsistent computations from the work queue.

It is possible to syntactically pre-compute pull-tab steps: that is, a case statement such as the one in (7) could implement pull-tabbing by defining an appropriate right-hand side rule for the choice branch. In fact, a major competing implementation of Curry does exactly that [8]. A disadvantage of that approach is that choice identifers must appear as first-class citizens of the program and be propagated through pull-tab steps using additional rules not encoded in the source program. We believe it is more efficient to embed choice identifiers in choice nodes as an implementation detail and process pull-tab steps dynamically. Section 4.2 compares these two approaches in greater detail.

4 Performance

In this section we present a set of benchmark results. These programs were previous used to compare three implementations of Curry [8]: Mcc, Pakcs, and KiCS2. We shall use KiCS2 to perform direct comparisons with Sprite[1], since it compares favorably to the others, and mention the relative performance of the others. KiCS2 compiles Curry to Haskell and then uses the Glasgow Haskell Compiler (GHC) [13] to produce executables. GHC has been shown to produce very efficient code [21, 20, 27]. Like Sprite, KiCS2 uses a pull-tabbing evaluation strategy, but unlike Sprite, it does not form a work queue; hence, is incomplete when faced with programs such as (10). Instead, it builds a tree containing all values of the program and executes (lazily and with interleaved steps) a user-selected search algorithm.

A major highlight of KiCS2 is that purely functional programs compile to "straight" Haskell, thus incurring no overhead due to the presence of unused logic capabilities.

[1] Available at https://github.com/andyjost/Sprite-3

Program	Type	KiCS2	Sprite	Δ
PaliFunPats	FL	0.64	0.09	-7.1
LastFunPats	FL	1.85	0.30	-6.2
Last	FL	1.90	0.31	-6.1
PermSortPeano	FL	44.04	8.14	-5.4
PermSort	FL	42.72	8.15	-5.3
ExpVarFunPats	FL	5.92	1.29	-4.6
Half	FL	42.31	9.55	-4.4
Reverse	F	0.36	0.21	-1.7
ReverseUser	F	0.34	0.21	-1.6
ReverseBuiltin	F	0.40	0.39	-1.0
ReverseHO	F	0.36	0.39	1.1
Primes	F	0.29	0.32	1.1
ShareNonDet	FL	0.28	0.33	1.2
PrimesBuiltin	F	0.73	1.10	1.5
PrimesPeano	F	0.41	0.66	1.6
QueensUser	F	0.87	1.83	2.1
Queens	F	0.80	1.81	2.3
TakPeano	F	0.84	2.08	2.5
Tak	F	0.32	0.92	2.9

Fig. 4: Execution times for a set of functional (F) and functional-logic (FL) programs taken from the KiCS2 benchmark suite. Times are in seconds. The final column (Δ) reports the speed-up (negative) or slow-down (positive) factor of Sprite relative to KiCS2. System configuration: Intel i5-3470 CPU at 3.20GHz, Ubuntu Linux 14.04.

Sprite, too, enjoys this zero-overhead property, but there is little room to improve upon GHC for functional programs, as it is the beneficiary of exponentially more effort. Our goal for functional programs, therefore, is simply to measure and minimize the penalty of running Sprite. For programs that utilize logic features KiCS2 emits Haskell code that simulates non-determinism. In these cases, there is more room for improvement since, for example, Sprite can avoid simulation overhead by more directly implementing logic features.

4.1 Functional Programs

The execution times for a set of programs taken from the KiCS2 benchmark suite[2] are shown in Fig. 4. The results are arranged in order from greatest improvement to greatest degradation in execution time. The most striking feature is the clear division between the functional (deterministic) and functional-logic (non-deterministic) subsets, which is consistent with our above-stated expectations. On average, Sprite produces relatively slower code for functional programs and relatively faster code for functional-logic ones. We calculate averages as the geometric mean, since that method is not strongly influenced by extreme results in either direction. The functional subset runs, on average, 1.4x slower in Sprite as compared to KiCS2. Figures published by Braßel et al. [8,

[2] Downloaded from https://www-ps.informatik.uni-kiel.de/kics2/benchmarks.

Fig.2, Fig.3] indicate that PAKCS and MCC run 148x and 9x slower than KICS2, respectively, for these programs. We take these results as an indication that the functional parts of SPRITE — i.e., those parts responsible for pattern-matching, rewriting, memory management, and optimization — although not as finely-tuned as their GHC counterparts, still compare favorably to most mainstream Curry compilers.

We note that SPRITE currently does not perform optimizations such as deforestation [14] or unboxing [21]. These, and other optimizations of ICurry, e.g., [6], could potentially impact the benchmark results. Inspecting the output of GHC reveals that the `tak` program (incidentally, the worse-case for SPRITE) is optimized by GHC to a fully-unboxed computation. To see how LLVM stacks up, we rewrote the program in C and converted it to LLVM using Clang [11], a C language front-end for LLVM. When we compiled this to native code and measured the execution time, we found that it was identical[3] to the KICS2 (and GHC) time. We therefore see no fundamental barrier to reducing the SPRITE "penalty" to zero for this program, and perhaps others, too. We have reason to be optimistic that implementing more optimizations at the source and ICurry levels, without fundamentally changing the core of SPRITE, will yield substantive improvements to SPRITE.

4.2 Functional-Logic Programs

For the functional-logic subset, Fig. 4 shows that SPRITE produces relatively faster code: 4.4x faster, on average. Published comparisons [8, Fig.4] indicate that, compared to KICS2, PAKCS is 5.5x slower and MCC is 3.5x *faster* for these programs. Our first thought after seeing this result was that SPRITE might enjoy a better algorithmic complexity. We had just completed work to reduce SPRITE's complexity when processing choices, so perhaps, we thought, in doing that work we had surpassed KICS2. We set out to test this by selecting a program dominated by choice generation and running it for different input sizes, with and without the recent modifications to SPRITE. The results are shown in Fig. 5. Contrary to our expectation, SPRITE and KICS2 exhibit strikingly similar complexity: both fit an exponential curve with r^2 in excess of 0.999, and their slope coefficients differ by less than 2%. A better explanation, then, for the difference is that some constant factor c exists, such that choice-involved steps in SPRITE are c-times faster than in KICS2. What could account for this factor? We believe the best explanation is the overhead of simulating non-determinism in Haskell, which we alluded to at the end of Sect. 3.3. To see why, we need to look at KICS2 in more detail.

KICS2 uses a few helper functions [8, Sect. 3.1] to generate choice identifiers:

```
thisID      :: IDSupply -> ID
leftSupply  :: IDSupply -> IDSupply
rightSupply :: IDSupply -> IDSupply
```
(12)

The purpose of these functions is to ensure that choice identifiers are never reused. Here, `ID` is the type of a choice identifier and `IDSupply` is opaque (for our purposes). Any function that might produce a choice is implicitly extended by KICS2 to accept a supply function. As an example, this program

[3] Using the Linux `time` command, whose resolution is 0.01 seconds.

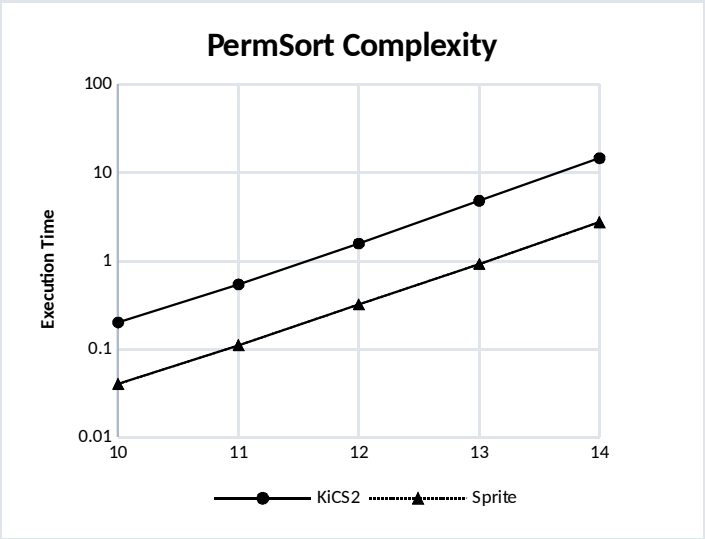

Fig. 5: Complexity analysis of `PermSort`. Execution times are shown for a range of problem sizes. The horizontal axis indicates the number of integers to sort by the permute-and-test method.

```
f :: Bool
main = xor f (False ? True)
```
(13)

is compiled to

```
main s = let s1 = leftSupply s
             s2 = rightSupply s
             s3 = leftSupply s2
             s4 = rightSupply s2
     in xor (f s3) (Choice (thisID s4) False True) s1
```
(14)

Clearly, the conversion to Haskell introduces overhead. The point here is simply to see that the compiled code involves five calls (to helper functions) that were not present in the source program. These reflect the cost of simulating non-determinism in a purely-functional language.

In Sprite, fresh choice identifiers are created by reading and incrementing a static integer. Compared to the above approach, fewer parameters are passed and fewer functions are called. A similar approach could be used in a Haskell implementation of Curry, but it would rely on impure features, adding another layer of complexity and perhaps interfering with optimizations. By contrast, the Sprite approach is extreme in its simplicity, as it executes only a few machine instructions. There is a remote possibility that a computation could exhaust the supply of identifiers since the type integer is finite. KiCS2 uses a list structure for choice identifiers and so does not suffer from this potential shortcoming. Certainly, the choice identifiers could be made arbitrarily large, but

doing so increases memory usage and overhead. A better approach, we believe, would be to compact the set of identifiers during garbage collection. The idea is that whenever a full collection occurs, Sprite would renumber the n choice identifiers in service at that time so that they fall into the contiguous range $0, \ldots, n-1$. This potential optimization illustrates the benefits of having total control over the implementation, since in this case it makes modifying the garbage collector a viable option.

5 Related Work

Several Curry compilers are easily accessible, most notably Pakcs [19], KiCS2 [8] and Mcc [25]. All these compilers implement a lazy evaluation strategy, based on definitional trees, that executes only needed steps, but differ in the control strategy that selects the order in which the alternatives of a choice are executed.

Both Pakcs and Mcc use backtracking. They attempt to evaluate all the values of the left alternative of a choice before turning to the right alternative. Backtracking is simple and relatively efficient, but incomplete. Hence, a benchmark against these compilers may be interesting to understand the differences between backtracking and pull-tabbing, but not to assess the efficiency of Sprite.

By contrast, KiCS2's control strategy uses pull-tabbing, hence the computations executed by KiCS2 are much closer to those of Sprite. KiCS2's compiler translates Curry source code into Haskell source code which is then processed by GHC [13], a mainstream Haskell compiler. The compiled code benefits from a variety of optimizations available in GHC. Section 4 contains a more detailed comparison between Sprite and KiCS2.

There exist other functional logic languages, e.g., $\mathcal{TOY}$ [10, 24], whose operational semantics can be abstracted by needed narrowing steps of a constructor-based graph rewriting system. Some of our ideas could be applied with almost no changes to the implementation of these languages.

A comparison with graph machines for functional languages is problematic at best. Despite the remarkable syntactic similarities, Curry's syntax extends Haskell's with a single construct, a free variable declaration, the semantic differences are profound. There are purely functional programs whose execution produces a result as Curry, but does not terminate as Haskell [5, Sect. 3]. Furthermore, functional logic computations must be prepared to encounter non-determinism and free variables. Hence, situations and goals significantly differ.

6 Future Work

Compilers are among the most complex software artifacts. They are often bundled with extensions and additions such as optimizers, profilers, tracers, debuggers, external libraries for application domains such as databases or graphical user interfaces. Given this reality, there are countless opportunities for future work. We have no plans at this time to choose any one of the extensions and additions listed above before any other. Some optimizations mentioned earlier, e.g., unboxing integers, are appealing only because they would improve some benchmark, and thus the overall perceived performance

of the compiler, but they may contribute very marginally to the efficiency of more realistic programs. Usability-related extensions and additions, such as aids for tracing and debugging an execution, and external libraries may better contribute to the acceptance of our work.

7 Conclusion

We have presented Sprite, a new native code compiler for Curry. Sprite combines the best features of existing Curry compilers. Similar to KiCS2, Sprite's strategy is based on pull-tabbing, hence there is no an inherent loss of completeness of compilers based on backtracking such as Pakcs and Mcc. Similar to Mcc, Sprite compiles to an imperative target language, hence is amenable to low-level machine optimization. Differently from all existing compilers, Sprite is designed to ensure operational completeness—all the values of an expression are eventually produced given enough computational resources.

Sprite's main intermediate language, ICurry, represents programs as graph rewriting systems. We described the implementation of Curry programs in imperative code using concepts of a low-level target language. Graph nodes are represented in memory as heap objects, and an efficient mechanism called tagged dispatch is used to perform pattern matches. Finally, we discussed the mechanisms used by Sprite to ensure completeness and consistency, and presented empirical data for a set of benchmarking programs. The benchmarks reveal that Sprite is competitive with a leading implementation of Curry.

References

1. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.
2. S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.
3. S. Antoy and M. Hanus. Declarative programming with function patterns. In *15th Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2005)*, pages 6–22, London, UK, September 2005. Springer LNCS 3901.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*, pages 87–101, Seattle, WA, August 2006. Springer LNCS 4079.
5. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
6. S. Antoy, J. Johannsen, and S. Libby. Needed computations shortcutting needed steps. In A. Middeldorp and F. van Raamsdonk, editors, Proceedings 8th International Workshop on *Computing with Terms and Graphs,* Vienna, Austria, July 13, 2014, volume 183 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–32. Open Publishing Association, 2015.
7. S. Antoy and A. Jost. Compiling a functional logic language: The fair scheme. In *23rd Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2013)*, pages 129–143, Madrid, Spain, Sept. 2013. Dpto. de Systems Informaticos y Computation, Universidad Complutense de Madrid, TR-11-13.

8. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
9. B. Brassel and F. Huch. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*, pages 122–138, Berlin, Heidelberg, 2007. Springer-Verlag.
10. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at http://toy.sourceforge.net.
11. *clang: a C language family frontend for LLVM*, 2016. Available at http://www.clang.llvm.org/.
12. R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz.
13. *The Glasgow Haskell Compiler*, 2013. Available at http://www.haskell.org/ghc/.
14. A. Gill, J. Launchbury, and S. L Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.
15. J. R. W. Glauert, R. Kennaway, G. A. Papadopoulos, and M. R. Sleep. Dactl: an experimental graph rewriting language. *J. Prog. Lang.*, 5(1):85–108, 1997.
16. M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*, 2006. Available at http://www-ps.informatik.uni-kiel.de/currywiki/.
17. M. Hanus. Flatcurry: An intermediate representation for Curry programs, 2008. Available at http://www.informatik.uni-kiel.de/ curry/flat/.
18. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
19. M. Hanus, editor. *PAKCS 1.11.4: The Portland Aachen Kiel Curry System*, 2014. Available at http://www.informatik.uni-kiel.de/ pakcs.
20. S. L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *Programming Languages and SystemsâĂŤESOP'96*, pages 18–44. Springer, 1996.
21. S. Peyton Jones and A. Santos. Compilation by transformation in the glasgow haskell compiler. In *Functional Programming, Glasgow 1994*, pages 184–204. Springer, 1995.
22. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO 04)*, pages 75–88, San Jose, CA, USA, Mar 2004.
23. F. J. López-Fraguas and J. de Dios-Castro. Extra variables can be eliminated from functional logic programs. *Electron. Notes Theor. Comput. Sci.*, 188:3–19, 2007.
24. F. J. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
25. W. Lux, editor. *The Muenster Curry Compiler*, 2012. Available at http://danae.uni-muenster.de/ lux/curry/.
26. S. Marlow and S. Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 4–15, New York, NY, USA, 2004. ACM.
27. W. Partain. The nofib benchmark suite of haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.
28. S. L Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201. ACM, 1989.