| TECHNICAL REPORT

Compiling Juvix to Cairo Assembly

Łukasz Czajka [a]

[a]Heliax AG

* *E-Mail: lukasz@heliax.dev*

Abstract

We describe a pipeline for compiling the functional programming language Juvix to the bytecode of the Cairo VM, which enables zero-knowledge proofs of Juvix program execution. The read-only memory model of Cairo fits well with the purely functional nature of Juvix, but also presents some unique challenges.

Keywords: Cairo ; Starknet ; Anoma ; Juvix ; Compilers ; zero-knowledge proofs ; functional programming

(Received: August 19, 2024; Version: September 10, 2024)

Contents

1 Introduction 2

2 Juvix 3

3 Cairo 4
- 3.1 Function calls . . . 7
- 3.2 Memory model . . . 8
- 3.3 Builtins . . . 9

4 Juvix to Cairo compilation pipeline 10
- 4.1 JuvixCore . . . 12
  - 4.1.1 Example programs . . . 13
  - 4.1.2 Stripped representation . . . 14
- 4.2 JuvixTree . . . 16
  - 4.2.1 Translation from Stripped JuvixCore . . . 19
  - 4.2.2 Compiling dynamic closure calls . . . 21
- 4.3 JuvixAsm . . . 22
  - 4.3.1 Translation from JuvixTree . . . 24
- 4.4 JuvixReg . . . 26
  - 4.4.1 Translation from JuvixAsm . . . 27
  - 4.4.2 Transformation into Static Single-Assignment form . . . 30

4.4.3 Optimization 31
4.4.4 Handling continuous memory 33
4.5 CASM 35
4.6 Cairo bytecode 41

5 Conclusion **41**

Acknowledgements **42**

References **43**

A Juvix CLI **44**

B Example program in different IRs **44**
B.1 Juvix 45
B.2 JuvixCore 45
B.3 Stripped JuvixCore 46
B.4 JuvixTree 46
B.5 JuvixAsm 47
B.6 JuvixReg 48
B.7 JuvixReg in SSA 49
B.8 Optimized JuvixReg 50
B.9 JuvixReg basic blocks 51
B.10 CASM 53

C CASM runtime **55**
C.1 Closure call 55
C.2 Closure extension 55

1. Introduction

Cairo [GPR21] is a practically-efficient Turing-complete STARK-friendly CPU architecture that allows generating zero-knowledge proofs of integrity for program execution. The Cairo Virtual Machine (Cairo VM) is used in Starknet – a ZK-rollup Layer 2 network that operates on top of Ethereum, enabling dApps to massively scale without compromising on security.

Juvix is an open-source functional programming language designed to write intent-centric privacy-preserving decentralised applications [Hel24] for the Anoma blockchain [GYB23]. The Juvix compiler pipeline allows relatively seamless

incorporation of different compilation targets. Currently, Cairo is used for private execution of Juvix programs.

This report describes the part of the Juvix compiler pipeline relevant to the Cairo backend. First, in Section 2 we discuss the Juvix language, its features and properties, as well as the basic architecture of the Juvix compiler. Section 3 presents the Cairo architecture, including its unique non-deterministic continuous read-only memory model. In Section 4 we detail the stages of the Juvix-to-Cairo compilation pipeline, including descriptions of intermediate representations (IRs) and translations between them. Finally, Section 5 summarizes the Juvix-to-Cairo pipeline, the encountered technical challenges and their solutions.

2. Juvix

Juvix is a purely functional programming language with eager (call-by-value) evaluation. As such, Juvix programs are referentially transparent mathematical functions without implicit state or any side effects. Juvix is a high-level language in the ML family (similar to, e.g., Haskell or OCaml) with many advanced features: algebraic data types, pattern matching, polymorphism, higher-order functions, traits, termination and positivity checking.

The Juvix compiler targets several different backends, including Cairo, Nockma (transparent VM for Anoma), RISC0 [BG23] and native code. After parsing, scoping and type checking, Juvix programs are desugared to JuvixCore – a minimalistic intermediate functional language [Cza23]. The relationship between Juvix and JuvixCore is similar to that between Haskell and Haskell Core. The compilation pipelines for different backends diverge after translation to JuvixCore.

Table 1 presents a feature comparison between Juvix, JuvixCore, Haskell and OCaml. In the case of JuvixCore, which does not specify a single type system, the "Yes" entries in the rows for polymorphism and data types mean that programs using these features can be directly represented in JuvixCore, not that type checking of such programs is performed by the current JuvixCore implementation. For more information on JuvixCore, see the report [Cza23].

Table 1. Comparison between language features supported by Juvix, JuvixCore, Haskell and OCaml.

Feature	Juvix	JuvixCore	Haskell	OCaml
Turing-complete	Yes[1]	Yes	Yes	Yes
Algebraic data types	Yes	Yes	Yes	Yes
GADTs	No	Yes	Yes	Yes
Prenex polymorphism	Yes	Yes	Yes	Yes
Higher-rank polymorphism	Some	Yes	Yes[2]	No
Hindley-Milner type inference	No	No	Yes	Yes
Type classes (traits)	Yes	No	Yes	No
Modules	Yes	No	Yes	Yes
Parameterised modules	No	No	No	Yes
Eager evaluation	Yes	Yes	Yes[3]	Yes
Lazy evaluation	No	No	Yes	Yes[4]
Metaprogramming	No	No	Yes[5]	Yes[6]

3. Cairo

The Cairo framework enables one to prove the integrity of an arbitrary computation. The Cairo VM executes Cairo bytecode creating an execution trace, which is later converted into an Algebraic Intermediate Representation (AIR) used to generate a zero-knowledge proof. The AIR encodes the steps of the computation as polynomial constraints.

Cairo Assembly (CASM) is a textual representation of Cairo bytecode. The instruction set follows the Reduced Instruction Set Computer (RISC) architecture. All instructions can be encoded using 15 flags and 3 integers. For a full description of CASM and the Cairo architecture, see [GPR21]. Here, we only give a brief overview and highlight the issues most relevant for compilation from a high-level purely functional language like Juvix.

The Cairo architecture was designed to allow a translation into an AIR with efficient zero-knowledge proof generation and verification. The design choices and performance considerations are therefore quite different from a conventional CPU architecture.

Cairo has random-access continuous read-only non-deterministic memory which stores elements of a certain finite field. The basic data type is thus a field: integers modulo a fixed large prime P. The memory is non-deterministically given by the prover at the start of the computation and cannot be modified later. Rather than as traditional loads and stores, Cairo memory access instructions

[1] via `terminating` and `positive` annotations

[2] with the `RankNTypes` extension.

[3] via strictness annotations.

[4] via the `Lazy.t` type.

[5] via Template Haskell

[6] via PPXs.

are better understood as asserting equalities between different memory cells. In case these assertions turn out inconsistent, execution in the Cairo VM fails. The execution of a Cairo program succeeds if there *exists* a memory assignment such that all assertions generated by the program instructions are satisfied.

Accessing memory in Cairo is not a performance bottleneck like in modern physical computers. Consequently, Cairo eschews general-purpose registers in favour of direct memory access. The notation $[n]$ is used to refer to the contents of the memory cell at address n.

In CASM, there are only three registers:

- `pc`: program counter. This register stores the address of the current instruction. It cannot be directly accessed in CASM. The program counter is increased after executing each non-branching instruction, and modified appropriately by jumps (`jmp`), calls (`call`) and returns (`ret`).
- `ap`: allocation pointer. This register points to the first free (unallocated) memory cell. It cannot be read directly. It can only be increased or used as an index in memory accesses, e.g., $[\texttt{ap} - 1]$ is a valid memory reference in CASM.
- `fp`: frame pointer. This register stores the address of the current function call frame. It cannot be read or written directly. It can only used as an index in memory accesses. The addresses of function arguments and local variables are relative to `fp`. When a function starts, `fp` is equal to `ap`. The value of `fp` doesn't change throughout the scope of a function, while `ap` increases on memory allocation.

In contrast to memory, the registers *are* mutable – their values can change as a result of executing an instruction.

An example CASM instruction is

$$[\texttt{ap}] = [\texttt{ap} - 1] * [\texttt{fp} + 2];\ \texttt{ap++}$$

which asserts that the memory cell at `ap` be equal to the memory cell at $\texttt{ap} - 1$ multiplied by the memory cell at $\texttt{fp} + 2$, and increases `ap` afterwards.

More precisely, the equality assertion instruction is

`<left> = <right>`

or

`<left> = <right>; ap++`

where `<left>` has the form $[r + k]$ and `<right>` has the form

- n – constant field element value, or
- $[r + k]$, or
- $[r_0 + k_0] \circ [r_1 + k_1]$, or
- $[r + k] \circ n$, or
- $[[r + k_0] + k_1]$,

where k, k_0, k_1 are 16-bit integer offsets, $r, r_0, r_1 \in \{\mathtt{ap}, \mathtt{fp}\}$, and $\circ \in \{+, *\}$.

Note that addition and field division can be "computed" thanks to non-determinism. For example, the following instruction "stores" $[\mathtt{ap} - 1] - [\mathtt{ap} - 2]$ in $[\mathtt{ap}]$ and increases ap:

$$[\mathtt{ap} - 1] = [\mathtt{ap}] + [\mathtt{ap} - 2];\ \mathtt{ap}{+}{+}$$

Such reshuffling is not necessary for subtraction by an immediate constant, which can simply be regarded as addition of a negative number, e.g., $[\mathtt{ap} - 2] - 1$ is syntactic sugar for $[\mathtt{ap} - 2] + (-1)$.

In addition to equality assertions, CASM supports the following instructions.

- `jmp` l: unconditional jump to label l.
- `jmp` l `if` $[r + k]$ `!=` 0: conditional jump to label l, where $r \in \{\mathtt{ap}, \mathtt{fp}\}$ and k is a 16-bit integer offset.
- `jmp rel <right>`: unconditional relative jump by `<right>` offset, where the form of `<right>` is the same as in equality assertions.
- `call` f: call the function f, where f is a label.
- `ret`: return from function call.
- `ap += n`: advance ap by n.

Below is an example CASM program which computes the factorial of 10. At the end of computation, the result is available in $[\mathtt{ap} - 1]$.

```
start:
  [ap] = 10
  [ap + 1] = 1
  ap += 2
loop:
  [ap] = [ap - 2] - 1
  [ap + 1] = [ap - 1] * [ap - 2]
  ap += 2
  jmp loop if [ap - 2] != 0
```

Note that each memory cell is assigned only once, and the assignments occur in order of increasing addresses. In each iteration of the loop, $[\mathtt{ap} - 2]$ is the

current loop counter (we count down from 10 to 0) and $[\mathtt{ap} - 1]$ is the product computed so far. Instead of overwriting two local variables, we use fresh copies of the variables in each iteration.

In the rest of this section, we discuss three aspects of Cairo relevant to compilation from a high-level functional language. We elaborate a bit more on the function call mechanism, the memory model, and Cairo builtins.

3.1. Function calls

Since Cairo memory is read-only, function calls cannot be implemented with a conventional call stack where frames are pushed on function call and popped on return. Instead, on each call a new frame is created (reserving the memory by advancing `ap`) which is never freed. In general, memory in Cairo is never freed, because it cannot be re-used due to the read-only nature of the memory model.

The `call` f instruction performs the following operations.

- Assert $[\mathtt{ap}] = \mathtt{fp}$, i.e., save the frame pointer.
- Assert $[\mathtt{ap} + 1] = \mathtt{pc}'$ where $\mathtt{pc}'$ points to the next instruction after the call, i.e., save the return address.
- `ap += 2`.
- Set $\mathtt{fp} = \mathtt{ap}$, i.e., make `fp` point to the new frame.
- Set $\mathtt{pc} = f$, i.e., jump to the label f.

Hence, at function entry $\mathtt{fp} = \mathtt{ap}$ and:

- $[\mathtt{fp} - 1]$ contains the return address,
- $[\mathtt{fp} - 2]$ contains the previous frame pointer,
- conventionally, $[\mathtt{fp} - 3]$ contains the first argument, $[\mathtt{fp} - 4]$ the second, and so on,

A function is called, e.g., like this:

```
[ap] = arg3; ap++
[ap] = arg2; ap++
[ap] = arg1; ap++
call f
```

The local variables are normally stored in $[\mathtt{fp}]$, $[\mathtt{fp} + 1]$, $[\mathtt{fp} + 2]$, etc.

The `ret` instruction performs the following operations.

- Set $\mathtt{pc} = [\mathtt{fp} - 1]$, i.e., jump to the return address saved by the `call` instruction.

- Set fp = [fp − 2], i.e., restore the previous frame pointer.

Conventionally, a function leaves its result in [ap − 1]. A function return looks like this:

```
[ap] = result; ap++
ret
```

To illustrate recursive function calls, below is a CASM program which computes the sum of numbers from 1 to 1000 using a recursive sum function. At the end of computation, the final result is in [ap − 1].

```
start:
  [ap] = 1000; ap++
  call sum
  jmp end
sum:
  jmp sum_label_1 if [fp - 3] != 0
  [ap] = 0; ap++
  ret
sum_label_1:
  [ap] = [fp - 3] - 1; ap++
  call sum
  [ap] = [fp - 3] + [ap - 1]; ap++
  ret
end:
```

The argument to sum (stored in [fp − 3]) is first compared against 0. In the zero case, the result is 0. In the non-zero case, sum is called recursively with the argument decreased by one, then the function returns the argument added to the recursive call result.

3.2. Memory model

We already briefly discussed aspects of the Cairo memory model. In this section, we elaborate on its crucial features and their relevance to compilation from a high-level functional language.

Cairo adopts a nondeterministic read-only continuous memory model. We explain each of these characteristics in turn.

- *nondeterministic*: memory is nondeterministically given by the prover at the start of the computation. Cairo instructions do not modify memory, but instead assert equalities between different memory cells.

- *read-only*: memory is read-only – it cannot be modified.
- *continuous*: memory accesses must occur with increasing addresses, leaving no "gaps" in between. For example, to access memory cell 100, one must first access all memory cells from 0 to 99 in order.

The read-only nature of Cairo memory fits well with the functional programming model where destructive updates are not permitted. The requirement of continuous memory access, on the other hand, causes significant complications, described in more detail in Section 4.4.4.

In [GPR21, Section 2.6], it is implied that memory accesses are checked for continuity only at the end, after the execution has finished. However, in the Rust implementation of Cairo VM [Lam24] the checks for memory continuity are performed periodically during the execution of the program, not only at the end. This complicates compilation. In particular, it is no longer possible to "reserve" space for local variables by increasing ap at function entry, and "fill in" the local variable values later.

For example, the following results in non-continuous memory access, because the access to [fp] in the third-last instruction occurs after accesses to higher memory addresses.

```
f:
  -- one local variable
  ap += 1
  -- now ap = fp + 1
  jmp lab if [fp - 3] != 0
  [fp] = 0
  ret
lab:
  [ap] = [fp - 3] - 1; ap++
  call f
  [fp] = [ap - 1] + 2
  [ap] = [fp] * 2; ap++
  ret
```

To avoid problems with memory access continuity checks, we require continuous memory access at every execution step.

3.3. Builtins

Cairo builtins [GPR21, Section 7] are predefined optimized low-level execution units in the Cairo VM. Communication with the builtins occurs via designated

memory addresses. For example, to use a hash builtin $H(x, y)$ which takes two arguments x and y, the user "writes" x and y at specified memory addresses m and $m+1$ (i.e., asserts appropriate equalities), and then "reads" $H(x, y)$ from the memory cell at address $m + 2$.

Currently, Juvix supports the following builtins.

- *Output*: specify program output.
- *Range Check*: verify that a value is in some bounded range $[0, n)$.
- *Elliptic Curve Operation*: compute $p + mq$ for points p, q on the STARK curve.
- *Poseidon Hash*: cryptographic hash designed to be efficient when expressed as an algebraic circuit.

4. Juvix to Cairo compilation pipeline

In this section, we describe the Juvix to Cairo compilation pipeline: the IRs and the translations between them. After scoping, parsing and type-checking, Juvix programs are desugared to JuvixCore where the pipelines for different backends begin to diverge. We describe only the backend part of the pipeline from JuvixCore to Cairo bytecode as it is most relevant to compilation to Cairo. Parts of the Cairo pipeline (up to the JuvixReg representation) are shared with different Juvix compiler targets.

The Juvix compiler backend pipeline architecture is presented schematically in Figure 1. The diagram nodes represent the IRs. The double arrows represent the transformations that are part of the Cairo pipeline. For completeness, we also show other Juvix compilation targets.

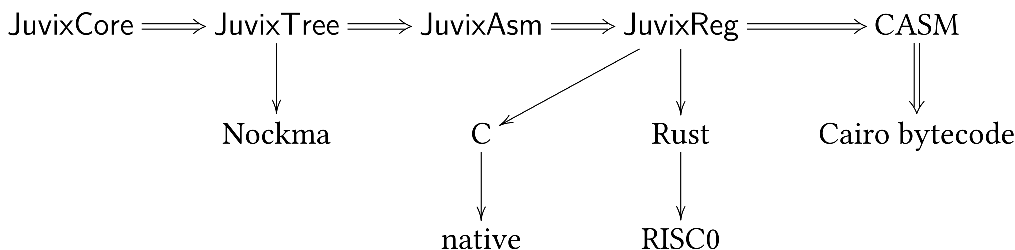

Figure 1. Juvix compiler backend pipeline.

First, we provide a brief overview of the IRs in the Cairo pipeline and the

translations between them. In the remaining subsections, we describe them in more detail.

1. JuvixCore is a minimalistic functional IR based on the lambda-calculus. The following major transformations are performed on JuvixCore to translate it to a Stripped JuvixCore representation.
    - *Eta-expansion* adjusts function arities.
    - *Pattern matching compilation* converts complex pattern matches into one-level case expressions.
    - *Lambda-lifting* removes anonymous and local functions.
    - *Optimization* performs inlining, specialization, constant folding and arithmetic simplification.
    - *Type erasure* removes runtime type information.
2. JuvixTree is an applicative functional IR with explicit closure operations and uncurried top-level functions. Translation from Stripped JuvixCore to JuvixTree involves the following.
    - *Application translation* selects the right JuvixTree operation for each JuvixCore application (direct call, static or dynamic closure call, closure allocation or extension, constructor data allocation).
    - *Dynamic closure call compilation* generates efficient code for call sites with possible partial application or overapplication.
3. JuvixAsm is a stack-based imperative assembly language suitable as an IR for eager purely functional languages. The translation to JuvixAsm linearizes JuvixTree expressions into sequences of stack-based instructions.
4. JuvixReg is a three-address code representation of JuvixAsm using local variables instead of the value stack. The following transformations are performed on JuvixReg before translating it to CASM.
    - *Static Single-Assignment form (SSA) transformation* ensures that each local variable is assigned only once, which is necessary because Cairo memory is read-only.
    - *Copy and constant propagation* removes spurious assignments.
    - *Basic block computation* with live variable analysis is a prerequisite for handling the continuity requirement of Cairo memory access.

4.1. JuvixCore

JuvixCore is a minimalistic intermediate functional language based on the lambda-calculus. Juvix desugars to JuvixCore, analogously to how Haskell desugars to Haskell Core. A detailed description of JuvixCore together with its precise evaluation semantics and optional type system is available in [Cza23]. Here, we provide only a brief and informal description, using the syntax of JuvixCore files (`*.jvc`) that can be parsed by JuvixCore-related CLI commands (see Appendix A).

A JuvixCore file consists of a semicolon-separated list of statements: type declarations and function definitions.

- *Inductive type declarations* have the form, e.g.,

  ```
  type list {
    nil : list;
    cons : Any -> list -> list;
  };
  ```

  This declares the type `list` with two constructors having the specified types. The predefined type `Any` is a universal type – any expression has this type.
- *Function definitions* have the form:

  ```
  def f := expr;
  ```

  or

  ```
  def f : type := expr;
  ```

  where `type` and `expr` are JuvixCore expressions. In JuvixCore, a function can have zero arguments.

A JuvixCore expression is one of the following.

- A variable, function, type or constructor identifier.
- A constant integer (e.g. `42`) or field element (e.g. `42F`).
- An application: `t s`.
- A lambda-expression: `\x t`.
- A let-expression: `let x := t in s`.
- A letrec-expression `letrec f := t in s`, or with multiple functions:

```
letrec[f g]
  f := t1;
  g := t2;
in s
```

This defines mutually recursive functions, where `f` and `g` can occur in `t1` and `t2`.

- A case-expression, e.g.:

```
case v of {
  c1 x1 x2 := b1;
  c2 x1 x2 x3 := b2
}
```

The case-expressions are one-level – nested patterns are not permitted. Above, the `x1`, `x2`, etc., are required to be variables binding the constructor arguments. In fact, JuvixCore also has match-expressions which allow complex nested patterns directly corresponding to Juvix pattern matches, but these are converted to case-expressions by the pattern matching compiler immediately after desugaring.
A case-expression matching on a boolean is presented in an `if-then-else` syntax:

```
if v then b1 else b2
```

is syntactic sugar for

```
case v of {
  true := b1;
  false := b2
}
```

- A primitive type: `Any`, `Int`, `Field`.
- A function type former: `T -> S` or `Pi A : Type, T`.

All binders (in lambda, let, letrec and case patterns) can have optional type annotations. Lack of type annotation is equivalent to the type being `Any`.

4.1.1. Example programs

As an example, we present a Juvix function computing the nth Fibonacci number in time $O(n)$.

```
fib : Nat -> Nat :=
  let
    fib' (x y : Nat) : Nat -> Nat
      | zero := x
      | (suc n) := fib' y (x + y) n;
  in fib' 0 1;
```

Below is the desugaring of this function to JuvixCore, with pattern matching already compiled and `Nat` converted to primitive integers.

```
def fib : Int -> Int :=
  letrec fib' : Int -> Int -> Int -> Int :=
    \(x : Int) \(y : Int) \(_X : Int)
      if = 0 _X then x else fib' y (+ x y) (- _X 1)
  in fib' 0 1;
```

Appendix B contains a bit more comprehensive example of the `partition` function in Juvix and all IRs discussed in this report.

4.1.2. Stripped representation

After desugaring Juvix to JuvixCore, the following transformations are performed before converting to JuvixTree.

1. *Eta-expansion*: adds lambda-abstractions at the top to make the number of function arguments match its type, e.g., expanding

   ```
   def f : Int -> Int := + 1
   ```

   to

   ```
   def f : Int -> Int := \(x : Int) + 1 x
   ```

2. *Pattern matching compilation*: compiles complex pattern matches into simple one-level case-expressions. The pattern matching compilation algorithm follows [Mar08].
3. *Primitive type conversion*: converts Juvix numeric types to JuvixCore primitive integers, with pattern matches converted to appropriate primitive arithmetic operations.
4. *Lambda lifting*: lifts out lambda-abstractions and local letrec-expressions into top-level definitions. For more information on lambda-lifting see [JL92, Chapter 6], [Jon87, Chapters 13,14] and [DS04].

5. *Optimization*: performs the following optimizations on the JuvixCore representation.

    - *Inlining* of functions explicitly marked for inlining or deemed small enough by a heuristic.

    - *Specialization* of functions for arguments explicitly marked for specialization. E.g., for the call `map id lst` a specialized version `map_id` of `map` is created with the `id` function substituted for the first argument of `map`.

    - *Constant folding*: evaluates fully applied non-recursive functions when all arguments are values, e.g., replacing $3 + 4$ with 7 and id 3 with 3.

    - *Case folding*: partially evaluates case expressions when the constructor matched on is known.

    - *Let folding*: folds lets whose values are immediate (variables or constants) or when the bound variable occurs at most once in the body. E.g., `let x := t in x + y` is replaced with `t + y`.

    - *Arithmetic simplification*: simplifies arithmetic operations, e.g., replacing $x + 0$ with x.

6. *Application sinking*: moves application arguments inside lets and cases, e.g., converting

    ```
    (let x := A in B) C
    ```

    into

    ```
    let x := A in B C
    ```

    and

    ```
    (case V of {P := A}) B
    ```

    into

    ```
    case V of {P := A B}
    ```

7. *Type erasure*: removes type arguments, type abstractions and type quantification.

After performing the above steps, we obtain a *Stripped* representation of JuvixCore. In this representation:

- all functions are defined at the top level (no local function definitions or lambda-abstractions except at the top),
- functions are simply-typed with the number of arguments (top lambda-abstractions) matching the type,
- all application expressions have the form $f t_1 \ldots t_n$ where f is a function name, a variable or a constructor.
- polymorphic arguments have the `Any` type.

Stripped JuvixCore can be directly translated to JuvixTree.

Example. Below is the Fibonacci function converted to Stripped JuvixCore.

```
def fib : Int -> Int := \(_X : Int) fib' 0 1 _X;
def fib' : Int -> Int -> Int -> Int :=
  \(x : Int) \(y : Int) \(_X : Int)
    if = 0 _X then x else fib' y (+ x y) (- _X 1);
```

4.2. JuvixTree

JuvixTree is an intermediate applicative functional language with explicit closure operations and uncurried top-level functions. There are no local function definitions or lambda-abstractions. JuvixTree supports type annotations with non-dependent function types ($A \to B$) and the universal type $*$ (this type is called `Any` in JuvixCore).

A JuvixTree program (in a `*.jvt` file) consists of a sequence of type and function definitions.

- *Inductive type declarations* have the form, e.g.,

  ```
  type list {
    nil : list;
    cons : (*, list) -> list;
  }
  ```

  The expression `(*, list) -> list` is the type of a function which takes two arguments, the first one of an arbitrary type, the second of type list, and returns a list. In JuvixTree, functions are uncurried and have a fixed total number of expected function arguments.
- *Function definitions* have the form, e.g.,

```
function f(arg1Type, arg2Type) : resultType {
  expr
}
```

where expr is a JuvixTree expression serving as the function body. Only the types of the arguments and the result need to be specified – argument names are optional. The nth argument can be referred to in the function body with `arg[n]` (where n is an integer constant).

Functions do not have local variables, but instead each has a *temporary stack* which can be extended with the save operation (explained below). The nth temporary stack cell from the bottom can be accessed with `tmp[n]`. Each function invocation creates a fresh temporary stack.

JuvixTree values are:

- constructor data $cv_1 \ldots v_n$ where c is a constructor name (or *tag*) and $v_1, \ldots, v_n$ are the values of the constructor arguments,
- closure $C(f, v_1, \ldots, v_k)$ where f is a function name, $v_1, \ldots, v_k$ are values of the first k function arguments, and $k < n$ with n the total number of function arguments,
- integers, booleans and field elements.

The values are stored in the temporary stacks and function arguments.

A JuvixTree expression is one of the following.

- A reference `arg[n]` to the nth function argument.
- A reference `tmp[n]` to the nth cell from the bottom in the temporary stack.
- A reference to a constructor field: `r.ctr[k]` where `r` is `arg[n]` or `tmp[n]`, and `ctr` is a constructor. This returns the kth constructor argument, assuming that the value at `r` is constructor data with tag `ctr`.
- A primitive operation, e.g., arithmetic operations on fields or integers. An important primitive operation is the unary `argsnum` which returns the number of arguments expected by a given closure (i.e., how many arguments still need to be supplied before a function call). In other words, `argsnum(t)` evaluates to $n - k$ if `t` evaluates to $C(f, v_1, \ldots, v_k)$ and n is the total number of arguments of f.
- An integer, field element or boolean constant.
- Constructor data allocation: `alloc[ctr](t1, ..., tn)` where `ctr` is a constructor tag and `t1`,...,`tn` are JuvixTree expressions.
- Closure allocation: `calloc[f](t1, ..., tn)`.

- Closure extension: `cextend(cl, t1, ..., tn)`. If `cl` evaluates to

$$C(f, v_1, \ldots, v_k)$$

  and `ti` evaluates to v'_i, and $k + n$ is smaller than the number of arguments of f, then the closure extension operation evaluates to

$$C(f, v_1, \ldots, v_k, v'_1, \ldots, v'_n).$$

- Function call: `call[f](t1, ..., tn)`. The number n of arguments supplied must be equal to the total number of arguments expected by `f`.
- Closure call: `call(cl, t1, ..., tn)`. The number n of arguments supplied must be equal to the remaining number of arguments expected by the closure `cl`.
- Dynamic closure call: `ccall(cl, t1, ..., tn)`. This operation implements the dynamic dispatch loop:

  1. it either calls or extends the closure `cl` depending on the number of supplied arguments versus the number of expected arguments fetched at runtime from the closure, and

  2. if the number of expected arguments is smaller than the number of supplied arguments, then the result of the call must be another closure and the process is repeated until we run out of supplied arguments.

- Save value on the temporary stack:

```
save(t) { expr }
```

  This operation evaluates `t` to v, pushes v on top of the temporary stack, evaluates `expr` to v', pops the temporary stack, and returns v'.
- Branch on a boolean:

```
br(t) {
  true: expr1
  false: expr2
}
```

- Branch on a constructor tag, e.g.:

```
case[list](lst) {
  nil: expr1
  cons: expr2
}
```

Each branch can be wrapped in `save { .. }` to indicate that the value matched on (e.g. `lst` above) should be saved (pushed) onto the temporary stack before evaluating the branch (and popped afterwards). E.g.:

```
case[list](lst) {
  nil: expr1
  cons: save {
    expr2
  }
}
```

4.2.1. Translation from Stripped JuvixCore

The special form of the Stripped representation of JuvixCore(see Section 4.1.2) lends itself well to a translation into JuvixTree. The JuvixTree operations are more detailed and low-level than the constructions available in Stripped JuvixCore. The gist of the translation is to choose the right JuvixTree operation for a given JuvixCore expression, depending on the context.

- Variables are translated into:
  - argument references (`arg[n]`) for variables representing function arguments, i.e., bound by the top lambda-abstractions,
  - temporary stack references (`tmp[n]`) for variables bound by let expressions,
  - constructor field references (`tmp[n].ctr[k]`) for variables bound by case expressions.
- Constants are translated into corresponding JuvixTree constants.
- Applications are translated into function calls, closure allocation, closure call, dynamic closure call, or constructor data allocation.
- Let-expressions are translated into `save` operations.
- Case-expressions are translated into branches on booleans or on constructor tags.

A more detailed explanations of some aspects of the translation follow.

Applications. A JuvixCore application $f\, t_1 \ldots t_n$ is translated to one of the following, where `ti` is the translation of t_i.

- `call[f](t1, ..., tn)` if f is a function name and n is the total number of arguments of f.
- `calloc[f](t1, ..., tn)` if f is a function name and n is smaller than the total number of arguments of f,
- `ccall(call[f](t1, ..., tk), tk+1, ..., tn)` if f is a function name and $n > k$ with k the total number of arguments of f,
- `ccall(r, t1, ..., tn)` if f is a variable and `r` is the corresponding reference in JuvixTree,
- `alloc(f, t1, ..., tn)` if f is a constructor name.

Case-expressions. A JuvixCore case-expression matching on an expression of type `I`, e.g.,

```
case a of {
  c1 x y := b1;
  c2 := b2;
}
```

is translated into

```
case[I](A) {
  c1: save {
    B1
  }
  c2: B2
}
```

where `A`, `B1`, `B2` are translations of `a`, `b1`, `b2` respectively, and x, y are translated inside `B1` into `tmp[k].c1[0]` and `tmp[k].c1[1]` respectively, where k is the index of the top of the temporary stack after the save for the `B1` branch.

Example. The Fibonacci function translated to JuvixTree is:

```
function fib(integer) : integer {
  call[fib'](0, 1, arg[0])
}
function fib'(integer, integer, integer) : integer {
  br(eq(0, arg[2])) {
```

```
    true: arg[0]
    false: call[fib'](arg[1], add(arg[0], arg[1]), sub(arg[2], 1))
  }
}
```

4.2.2. Compiling dynamic closure calls

After translating Stripped JuvixCore to JuvixTree, the next step is to remove dynamic closure calls. The `ccall` operation has a more complex semantics than other JuvixTree operations. For a fixed constant number n of supplied arguments, it can be implemented as a function `apply_n` in JuvixTree without `ccall`. As an example, below is a JuvixTree implementation of `apply_3`.

```
function apply_3(*, *, *, *) : * {
  save(argsnum(arg[0])) {
    br(eq(3, tmp[0])) {
      true: call(arg[0], arg[1], arg[2], arg[3])
      false: br(eq(2, tmp[0])) {
        true: call[apply_1](call(arg[0], arg[1], arg[2]), arg[3])
        false: br(eq(1, tmp[0])) {
          true: call[apply_2](call(arg[0], arg[1]), arg[2], arg[3])
          false: cextend(arg[0], arg[1], arg[2], arg[3])
        }
      }
    }
  }
}
```

The first argument of `apply_n` is the closure to be called or extended. The remaining arguments are the arguments supplied to the dynamic closure call. First, we compute the number of arguments k expected by the closure (`argsnum(arg[0])`), which we then compare against the number n of supplied arguments.

- If $k = n$ then we call the closure.
- If $k < n$ then we compute

  ```
  call[apply_{n-k}]
  (
    call(arg[0], arg[1], ..., arg[k]),
    arg[k+1], ..., arg[n]
  )
  ```

i.e., we call the closure with k arguments, and then pass the result to `apply_{n-k}` with the remaining $n - k$ arguments.

- If $k > n$ we extend the closure with the n supplied arguments.

One can generate `apply_n` functions for arbitrary $n > 0$. Then

```
ccall(cl, t1, ..., tn)
```

is translated to

```
call[apply_n](cl, t1, ..., tn)
```

In this way, we solve the problem of compiling partial application and overapplication in a curried functional programming language, without introducting a general dynamic dispatch loop in the runtime. Surprisingly, it is difficult to find a clear and accessible description of this simple technique in the functional language compiler literature. This method for curried function application is essentially a variant of "eval/apply" where the caller is responsible for arity matching. An analogous technique is used in the native-code OCaml compiler. The presentation [Ler05] gives a good explanation of curried function application compilation, why "eval/apply" is desirable for native compilation, and how the native-code OCaml compilation grew out of different abstract machines. See also [MJ06] for a comparison of "eval/apply" with "push/enter" and evidence supporting the use of "eval/apply" in compiled implementations.

4.3. JuvixAsm

JuvixAsm is a stack-based imperative assembly language well-suited as an IR for a strongly typed purely functional language with eager evaluation. In contrast to most assembly languages, JuvixAsm abstracts aways memory management. It has only allocation instructions (for closures and constructor data) but no deallocation, garbage-collection control or other memory management instructions. JuvixAsm can be seen as an assembly representation of JuvixTree, with JuvixAsm instructions more detailed and low-level versions of JuvixTree operations.

A JuvixAsm program (in a `*.jva` file) is a sequence of type and function definitions. The type declarations and function definition headings have the same syntax as in JuvixTree – only function bodies are now semicolon-seprated lists of JuvixAsm instructions instead of JuvixTree expressions.

In JuvixAsm, there is a global *call stack* of *call frames*. Every function invocation has a separate call frame containing the following.

- *Argument area*: stores function arguments. The nth argument is referred to with `arg[n]` like in JuvixTree.

- *Termorary stack*: stores temporary values, analogous to JuvixTree temporary stack. The *n*th temporary stack cell from the bottom is referred to with `tmp[n]`.
- *Value stack*: stores intermediate computation results. JuvixAsm instructions typically pop their arguments from the value stack and push the result on the top.

In what follows, when referring to the stack we mean the value stack for the current function invocation (call frame), unless otherwise stated.

The values stored in memory are the same as in JuvixTree: constructor data, closures, integers, booleans and field elements. A JuvixAsm value reference is one of:

- an integer, boolean or field element constant,
- `arg[n]` or `tmp[n]`,
- `arg[n].ctr[k]` or `tmp[n].ctr[k]`.

A JuvixAsm instruction is one of the following.

- `push r` pushes onto the stack the value referred to by the JuvixAsm reference `r`.
- `pop` pops the stack.
- A primitive operation instruction corresponds to a primitive operation in JuvixTree. It pops its arguments from the stack, performs the operation and pushes the result.
- `alloc c` allocates constructor data with tag *c* and arguments popped from the stack. The result is pushed on the stack.
- `calloc f n` allocates a closure for the function *f* with *n* supplied arguments popped from the stack. The resulting closure is pushed on the stack.
- `cextend n` pops a closure from the stack and extends it with *n* arguments popped from the stack in order. The result is then pushed on the stack.
- `call f` calls the function *f* with *n* arguments popped from the stack, where *n* is the total number of arguments expected by *f*. A new call frame for *f* is pushed onto the global call stack and the arguments are transferred into its argument aread. After the call finishes, the result is pushed on top of the value stack.
- `tcall f` tail-calls the function *f*. This is the same as `call f`, except that instead of pushing a new call frame onto the global call stack, the current call frame ie reused. The called function *f* then returns to the caller of the current function.

- `call $ n` pops a closure from the stack and calls it with n arguments popped from the stack in order. The number of supplied arguments n must match the number of arguments expected by the closure.
- `tcall $ n` tail-calls a closure on top of the stack with n supplied arguments.
- `ret` returns from the current function: pops the global call stack and returns to the caller. The result of the call is taken from the top of the value stack of the callee and pushed on top of the value stack of the caller.
- Branch on a boolean value on top of the stack. Pops the stack.

```
br {
  true: {..};
  false: {..};
}
```

- Branch on the tag of the constructor data on top of the stack. Does *not* pop the stack.

```
case I {
  c1: {..};
  ..
  cn: {..};
}
```

  where `I` is the inductive type name.
- Save the top of the value stack on the temporary stack.

```
save {..}
```

  This instruction pushes the top of the value stack onto the temporary stack, pops the value stack, executes the nested code, and pops the temporary stack afterwards.
- Tail save: `tsave {..}`. This is the same as `save` except that it does not pop the temporary stack after executing the nested code. Typically, the nested code returns from the current function.

4.3.1. Translation from JuvixTree

JuvixTree operations are translated to corresponding JuvixAsm instructions using the value stack to store intermediate results. For example, the JuvixTree expression

```
add(2, call[f](tmp[1].ctr[1], mul(arg[0], 3)))
```

is translated to a sequence of JuvixAsm instructions

```
push 3;
push arg[0];
mul;
push tmp[1].ctr[1];
call f;
push 2;
add;
```

Executing a sequence of JuvixAsm instructions corresponding to a JuvixTree expression results in pushing the value of the expression on top of the value stack.

In general, let $\mathcal{T}(t)$ be a sequence of JuvixAsm instructions for the JuvixTree expression t. Then for a JuvixTree operation o applied to JuvixTree expressions $t_1, \ldots, t_n$ we define:

$$\mathcal{T}(o(t_1, \ldots, t_n)) = \mathcal{T}(t_n) \ldots \mathcal{T}(t_1)\mathcal{T}(o)$$

i.e., we concatenate the lists of instructions generated for the arguments and append the instruction $\mathcal{T}(o)$ corresponding to the operation o. When executed, each $\mathcal{T}(t_i)$ pushes the value v_i of t_i on the stack. Then $\mathcal{T}(o)$ pops the arguments $v_1, \ldots, v_n$ from the stack, executes the corresponding operation, and pushes the result $o(v_1, \ldots, v_n)$ on top of the stack. This is a standard method for compiling expressions into a stack machine [CT23, Section 4.4.1].

For each JuvixTree expression e, the translation keeps track of whether e is a tail expression, i.e., if e is returned as the result of the function without further processing. For example, in

```
function fib'(integer, integer, integer) : integer {
  br(eq(0, arg[2])) {
    true: arg[0]
    false: call[fib'](arg[1], add(arg[0], arg[1]), sub(arg[2], 1))
  }
}
```

the `br(..) {..}` and its branches (the `arg[0]` after `true:` and the `call[fib'](..)`) are the only tail expressions.

Tail calls and saves are translated into `tcall` and `tsave` instructions. In order to return the value of a tail expression as a result of the function, the `ret` instruction is appended after non-tail non-branching JuvixAsm instructions generated for JuvixTree tail expressions (i.e., except after `br`, `case`, `tcall` and `tsave`).

Example. The Fibonacci function translated to JuvixAsm follows.

```
function fib(integer) : integer {
  push arg[0];
  push 1;
  push 0;
  tcall fib';
}
function fib'(integer, integer, integer) : integer {
  push arg[2];
  push 0;
  eq;
  br {
    true: {
      push arg[0];
      ret;
    };
    false: {
      push 1;
      push arg[2];
      sub;
      push arg[1];
      push arg[0];
      add;
      push arg[1];
      tcall fib';
    };
  };
}
```

4.4. JuvixReg

JuvixReg is a register-based imperative assembly language designed as an IR for eager purely functional programming languages. JuvixReg can be seen as a three-address code [CT23, Section 4.4.2],[ALSU07, Section 6.2] rendering of

JuvixAsm. Like JuvixAsm, it has no explicit memory management. The JuvixReg instructions correspond to JuvixAsm instructions.

A JuvixReg program (in a `*.jvr` file) is a sequence of type and function definitions. The syntax of type declarations and function headings is the same as in JuvixAsm and JuvixTree, only function bodies are sequences of semicolon-separated JuvixReg instructions.

Instead of the JuvixAsm value and temporary stacks, each JuvixReg function has a finite number of temporary variables ("registers") which can be assigned multiple times. The *n*th temporary variable is referred to with `tmp[n]`. A JuvixReg value reference `r` has the same syntax as a JuvixAsm value reference, only now `tmp[n]` denotes a JuvixReg temporary variable instead of a JuvixAsm temporary stack cell.

A JuvixReg instruction is one of the following.

- Assignment: `tmp[n] = r`.
- Primitive operation: `tmp[n] = op r` or `tmp[n] = op r1 r2`.
- Constructor data allocation: `tmp[n] = alloc c (r1, .., rn)`.
- Closure allocation: `tmp[n] = calloc f (r1, .., rn)`.
- Closure extension: `tmp[n] = cextend r (r1, .., rn)`.
- Call: `tmp[n] = call f (r1, .., rn)` where `f` is a function name or a closure value reference.
- Tail call: `tcall f (r1, .., rn)` where `f` is a function name or a closure value reference.
- Return: `ret r`.
- Boolean branch: `br r { true: {..}; false: {..}; }`.
- Case: `case r { c1: {..}; ..; cn: {..}; }`.

The *branching instructions* `br` and `case` can optionally specify the *output variable*:

- `br r, out: tmp[n] { true: {..}; false: {..}; }`,
- `case r, out: tmp[n] { c1: {..}; ..; cn: {..}; }`.

The output variable is a temporary variable which stores the result of the computation in each branch. In case the output variable is specified, it is the only variable assigned in the branches that can be read in subsequent instructions after the branching instruction.

4.4.1. Translation from JuvixAsm

Since a single JuvixAsm function contains no loops, the height of the value stack can be determined at compilation time for each instruction in the function body.

We assume that different branches of `br` and `case` instructions are consistent in their stack manipulations, i.e., they all decrease or increase the stack height by the same amount. This assumption, however, is satisfied for any code generated from JuvixTree expressions. In fact, for non-tail branching instructions the stack height is increased by exactly one – the result of branch evaluation is pushed on the stack.

The translation from JuvixAsm to JuvixReg simply assigns consecutive temporary variables to JuvixAsm value stack cells. Since we know the current value stack height at each instruction, we know which temporary variable to assign when pushing the stack, and which ones to read when popping it. For example, the JuvixAsm `fib'` function (see the end of the previous section) is translated into:

```
function fib'(integer, integer, integer) : integer {
  tmp[0] = arg[2];
  tmp[1] = 0;
  tmp[0] = eq tmp[1] tmp[0];
  br tmp[0] {
    true: {
      tmp[0] = arg[0];
      ret tmp[0];
    };
    false: {
      tmp[0] = 1;
      tmp[1] = arg[2];
      tmp[0] = sub tmp[1] tmp[0];
      tmp[1] = arg[1];
      tmp[2] = arg[0];
      tmp[1] = add tmp[2] tmp[1];
      tmp[2] = arg[1];
      tcall fib' (tmp[2], tmp[1], tmp[0]);
    };
  };
}
```

The variable `tmp[n]` corresponds to the nth value stack cell from the bottom.

Similarly, the height of the temporary stack in a JuvixAsm function can be determined at compilation time. If the maximum temporary stack height is h, then JuvixReg temporary variables `tmp[0]` to `tmp[h-1]` are reserved for tempo-

rary stack cells. The variable `tmp[h + n]` is reserved for the *n*th value stack cell.

Recall that each branch in a non-tail JuvixAsm `br` or `case` pushes the branch computation result onto the value stack, increasing its height by exactly one. Hence, the corresponding JuvixReg non-tail branching instruction can be unambiguously associated an output variable – the temporary variable corresponding to the top of the stack at the end of each branch. For example, the following JuvixAsm function

```
function foo(integer, integer) : integer {
  push arg[0];
  push arg[1];
  eq;
  br {
    true: {
      push 1;
      push arg[1];
      add;
    };
    false: { push 2; };
  };
  push arg[0];
  add;
  ret;
}
```

is translated to

```
function foo(integer, integer) : integer {
  tmp[0] = arg[0];
  tmp[1] = arg[1];
  tmp[0] = eq tmp[1] tmp[0];
  br tmp[0], out: tmp[0] {
    true: {
      tmp[0] = 1;
      tmp[1] = arg[1];
      tmp[0] = add tmp[1] tmp[0];
    };
    false: {
      tmp[0] = 2;
```

```
    };
  };
  tmp[1] = arg[0];
  tmp[0] = add tmp[1] tmp[0];
  ret tmp[0];
}
```

4.4.2. Transformation into Static Single-Assignment form

The idea of compiling JuvixReg to CASM is to use a fixed memory location relative to the Cairo frame pointer ([fp + k]) for each JuvixReg temporary variable. Since CASM memory is read-only, we cannot have more than one "assignment" to the same memory location. To ensure that, we first transform JuvixReg into Static Single-Assignment form (SSA) [CT23, Section 9.3],[App98] where each variable is assigned only once. For example, the result of translating the fib' function into SSA is:

```
function fib'(integer, integer, integer) : integer {
  tmp[0] = arg[2];
  tmp[1] = 0;
  tmp[2] = eq tmp[1] tmp[0];
  br tmp[2] {
    true: {
      tmp[3] = arg[0];
      ret tmp[3];
    };
    false: {
      tmp[3] = 1;
      tmp[4] = arg[2];
      tmp[5] = sub tmp[4] tmp[3];
      tmp[6] = arg[1];
      tmp[7] = arg[0];
      tmp[8] = add tmp[7] tmp[6];
      tmp[9] = arg[1];
      tcall fib' (tmp[9], tmp[8], tmp[5]);
    };
  };
}
```

In contrast to standard SSA, we allow the same variable to be assigned in different branches of a branching instruction. The assigned variables are required to

be unique only within a single execution path.

Since we compute SSA for each function separately and JuvixReg functions do not contain loops, a naive SSA computation algorithm is sufficient. Instead of using the ϕ-functions [CT23, Section 9.3], when necessary we insert extra assignments to ensure that each branch in a branching instruction has the same output variable. For example, here is the `foo` function from above in SSA:

```
function foo(integer, integer) : integer {
  tmp[0] = arg[0];
  tmp[1] = arg[1];
  tmp[2] = eq tmp[1] tmp[0];
  br tmp[2], out: tmp[6] {
    true: {
      tmp[3] = 1;
      tmp[4] = arg[1];
      tmp[5] = add tmp[4] tmp[3];
      tmp[6] = tmp[5];
    };
    false: {
      tmp[3] = 2;
      tmp[6] = tmp[3];
    };
  };
  tmp[7] = arg[0];
  tmp[8] = add tmp[7] tmp[6];
  ret tmp[8];
}
```

The assignments to `tmp[6]` were inserted to “unify” the output variables for both branches.

4.4.3. Optimization

The translation from JuvixAsm to JuvixReg generates many unnecessary assignments. An assignment is generated for any stack push, including for constants and references to function arguments or temporary stack cells. The spurious assignments are removed by the following two optimizations.

1. *Constant and copy propagation.* Assignments of constants (`tmp[n] = c`) and variable references (`tmp[n] = tmp[k]` and `tmp[n] = arg[k]`) are propagated to subsequent instructions, i.e., all succeeding occurrences of `tmp[n]`

are replaced with the right-hand side of the assignment. Arithmetic operations with known constant arguments are evaluated and propagated further. Branching on known constant values is simplified.

2. *Dead assignment elimination.* Assignments to dead variables are removed. A variable is dead if it is not read in any subsequent instructions.

Below is the result of running copy and constant propagation on the `fib'` function, followed by dead assignment elimination. We renumber the temporary variables left after removing dead assignments.

```
function fib'(integer, integer, integer) : integer {
  tmp[0] = eq 0 arg[2];
  br tmp[0] {
    true: {
      ret arg[0];
    };
    false: {
      tmp[1] = sub arg[2] 1;
      tmp[2] = add arg[0] arg[1];
      tcall fib' (arg[1], tmp[2], tmp[1]);
    };
  };
}
```

The optimized version of the `foo` function is:

```
function foo(integer, integer) : integer {
  tmp[0] = eq arg[1] arg[0];
  br tmp[0], out: tmp[1] {
    true: {
      tmp[1] = add arg[1] 1;
    };
    false: {
      tmp[1] = 2;
    };
  };
  tmp[2] = add arg[0] tmp[1];
  ret tmp[2];
}
```

4.4.4. Handling continuous memory

A major issue in the translation of JuvixReg to CASM is the requirement that Cairo memory accesses have to be continuous. Recall from Section 3.2 that memory accesses are continuous if they occur in order of increasing addresses with no gaps. The continuity requirement makes it impossible to use a simple standard method for compiling local variables: reserve space for k variables at function entry by increasing `ap` by k and translate `tmp[n]` to [`fp` + n]. There is no guarantee that the variables are accessed in an increasing order.

In general, it may be impossible to reorder the variables in such a way that memory accesses become continuous. The problem is that in, e.g., a call instruction an undetermined amount of memory can be accessed, with the next access address increasing by an offset that cannot be determined at compilation time. However, it is not difficult to assign continuous memory addresses to variables within a single *basic block* [ALSU07, Section 8.4] – a maximal sequence of consecutive instructions with no jumps or unbounded dynamic allocation.

This suggests the following approach to translating local temporary variables. We divide the JuvixReg instruction sequence for a function body into basic blocks. In this context, a basic block is a sequence of instructions ending with one of: `cextend`, `call`, `tcall`, `ret`, `br`, `case`. A basic block contains only one of these instructions at the end – the *final instruction*. The branches of `br` and `case` constitute separate basic blocks.

The basic blocks form the nodes of the control-flow graph (CFG) with edges denoting transitions to other basic blocks. The edges are labeled to indicate whether the transition occurs after executing the final instruction or in a branch of the final instruction. Below we present the CFG for the `foo` function. The basic blocks are denoted by `block B {..}`. The transition edges are denoted by `goto`.

```
block B1 {
  tmp[0] = eq arg[1] arg[0];
  br tmp[0], out: tmp[1] {
    true: {
      goto B2;
    };
    false: {
      goto B3;
    };
  };
}
```

```
block B2 {
  tmp[1] = add arg[1] 1;
  goto B4;
}
block B3 {
  tmp[1] = 2;
  goto B4;
}
block B4 {
  tmp[2] = add arg[0] tmp[1];
  ret tmp[2];
}
```

The next step is to compute the set of variables *live* [ALSU07, Section 8.4.2] at the beginning of each basic block, i.e., variables that are used subsequently in the function (including other basic blocks) without being reassigned first. The variables live at the start of the blocks of `foo` are as follows:

- `B1`: `arg[0]`, `arg[1]`,
- `B2`: `arg[0]`, `arg[1]`,
- `B3`: `arg[0]`,
- `B4`: `arg[0]`, `tmp[1]`.

As another example, the basic blocks for the `fib'` function are presented below, with the live variables indicated.

```
block B1, live: (arg[0], arg[1], arg[2]) {
  tmp[0] = eq 0 arg[2];
  br tmp[0] {
    true: { goto B2; };
    false: { goto B3; };
  };
}
block B2, live: (arg[0]) {
  ret arg[0];
}
block B3, live: (arg[0], arg[1], arg[2]) {
  tmp[1] = sub arg[2] 1;
  tmp[2] = add arg[0] arg[1];
  tcall fib' (arg[1], tmp[2], tmp[1]);
}
```

Within a single basic block, we know at each instruction how many variables have been assigned before and can thus uniquely associate an offset k to the kth assigned variable. References to the variable are then translated to $[\mathtt{fp} + k]$. Transitions to other basic blocks are compiled to calls transferring live variables as arguments. The translation of JuvixReg into CASM is described in more detail in the next section.

4.5. CASM

We translate each JuvixReg basic block to CASM separately. Within a single basic block, at each instruction we know how many variables have already been assigned. More generally, we know the `ap`-*offset* – by how much `ap` increased since the beginning of the basic block. If the instruction assigns its result to `tmp[n]` and the current `ap`-offset is k, we generate an appropriate CASM equality assertion $[\mathtt{ap}] = R$; `ap++` and associate `tmp[n]` with $[\mathtt{fp} + k]$, i.e., we set the `fp`-*offset* of `tmp[n]` to k. Recall that at function entry `fp = ap`, so $[\mathtt{fp} + k]$ refers to the memory cell "assigned" in the generated equality assertion. Because the JuvixReg code is in SSA, we will associate an `fp`-offset to a given variable only once. A transition into another basic block B at the final instruction is translated into a call of the block B, with the variables live at the start of B transferred as arguments. The `call` instruction sets `fp = ap`, which essentially "resets" the `ap`- and `fp`-offsets back to `0`, enabling their static calculation in the next basic block B.

Hence, each basic block is effectively treated as a separate function. The basic blocks for the branches of branching instructions are an exception. For these blocks a simple jump suffices because when entering the branch we still know the `ap`-offset. In general, we do not know the `ap`-offset after executing a branch, because `ap` might change differently in different branches.

We use the calling convention from Section 3.1 with one modification. Recall that at function entry the n function arguments are available in $[\mathtt{fp} - 3]$, ..., $[\mathtt{fp} - 2 - n]$. One extra $n + 1$-th argument is automatically passed to each function: the *builtins pointer*. This argument is then available in $[\mathtt{fp} - 3 - n]$. Recall from Section 3.3 that each Cairo builtin has a designated memory address used for communication with the builtin. The builtins pointer points to a structure consisting of the memory addresses associated with the supported builtins. These addresses change when using the builtins. The new builtins pointer is returned by each function in $[\mathtt{ap} - 2]$. Recall that the function result is returned in $[\mathtt{ap} - 1]$.

In our internal representation of CASM, we use extra arithmetic and field op-

erations which are later translated to appropriate equality assertion instructions in the Cairo bytecode. The idea is to indicate which side of an equality assertion is the destination that is "assigned" by the instruction, which allows for conventional execution of generated CASM code. For example, we have an extra integer subtraction instruction `isub`. Then, e.g.,

```
[ap] = [ap - 1] isub [ap - 2]; ap++
```

is compiled as

```
[ap - 1] = [ap] + [ap - 2]; ap++
```

By using `isub` we indicate that `[ap]` is the destination of the "assignment". Currently, no bound checks are generated for integer arithmetic, but this may change in the future.

Below is annotated CASM code generated for the `foo` function.

```
foo:
  -- block B1
  -- tmp[0] = eq arg[1] arg[0]
  -- true is zero, false is non-zero
  [ap] = [fp - 4] - [fp - 3]; ap++
  -- [fp] is tmp[0]
  jmp label_11 if [fp] != 0
  -- block B2
  -- tmp[1] = add arg[1] 1
  [ap] = [fp - 4] iadd 1; ap++
  -- [fp + 1] is tmp[1]
  -- goto B4
  -- [fp - 5] is builtins pointer, passed as the last argument
  [ap] = [fp - 5]; ap++
  [ap] = [fp - 3]; ap++
  [ap] = [fp + 1]; ap++
  call rel 3
  ret
  jmp label_12
  -- block B3
label_11:
  -- tmp[1] = 2
  [ap] = 2; ap++
```

```
  -- [fp + 1] is tmp[1]
  -- goto B4
  [ap] = [fp - 5]; ap++
  [ap] = [fp - 3]; ap++
  [ap] = [fp + 1]; ap++
  call rel 3
  ret
  -- block B4
label_12:
  -- tmp[2] = add arg[0] tmp[1]
  [ap] = [fp - 4] iadd [fp - 3]; ap++
  -- [fp] is tmp[2]
  -- return the builtins pointer
  [ap] = [fp - 5]; ap++
  -- return the result: tmp[2]
  [ap] = [fp]; ap++
  ret
```

Notice that program control transitions to blocks `B2` and `B3` directly or via simple jumps, because these blocks correspond to the branches in `br`. The transitions to block `B4` are via a relative call. The instructions

```
call rel 3
ret
```

first call the code beginning after the `ret`. When that code returns, the current function executes the `ret` and returns with the same result. The relative offset 3 indicates the number of memory cells (not instructions) to jump forward for the call. The `call 3` instruction occupies two memory cells in the Cairo bytecode, `ret` occupies one.

As another example, here is the `fib'` function in CASM.

```
fib':
  -- block B1
  -- [fp - 5] is arg[2]
  jmp label_12 if [fp - 5] != 0
  -- block B2
  -- [fp - 6] is the builtins pointer
  [ap] = [fp - 6]; ap++
  [ap] = [fp - 3]; ap++
```

```
  ret
label_12:
  -- block B3
  -- tmp[1] = sub arg[2] 1
  [ap] = [fp - 5] isub 1; ap++
  -- [fp] is tmp[1]
  -- tmp[2] = add arg[0] arg[1]
  [ap] = [fp - 3] iadd [fp - 4]; ap++
  -- [fp + 1] is tmp[2]
  -- [fp - 6] is the builtins pointer
  [ap] = [fp - 6]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp + 1]; ap++
  [ap] = [fp - 4]; ap++
  call fib'
  ret
```

Note that both basic blocks B2 and B3 are transitioned to without a call. Both blocks correspond to branches. The tail call to fib' in the second branch is compiled to call followed by ret. In CASM, there is no way to replace the current frame, so tail calls must create new frames.

Below we discuss several more detailed issues related to the translation from JuvixReg to CASM.

Constructors. Memory layout for non-record constructor data is one memory cell for the constructor id (CID) followed by constructor argument representations (each a pointer or a field element taking up one memory cell). The CID is equal to $2i+1$ where i is the 0-based index of the constructor within its inductive type. Such an encoding enables simpler compilation of case switches (see next paragraph). For record constructors, the CID is omitted.

The memory for constructor data is allocated at [ap] with ap increased afterwards appropriately. Since the size of the constructor (its number of arguments and if it's a record constructor) is a constant known at compilation time, the new ap-offset can be computed and no transition to a new basic block is necessary. Unfortunately, in CASM it is not possible to access the value of ap directly, which we need in order to store a pointer to allocated data. The following function juvix_get_regs, recommended in [GPR21, Section 8.4], allows to obtain the value of ap.

```
juvix_get_regs:
```

```
  call juvix_get_ap_reg
  ret
juvix_get_ap_reg:
  ret
```

After calling `juvix_get_regs` we have $[\mathsf{ap}-2] = \mathsf{ap}-2$. The JuvixReg allocation instruction

```
r = alloc c (r1, .., rn)
```

is compiled to

```
  call juvix_get_regs
  -- set the CID
  [ap] = CID; ap++
  -- set the arguments
  [ap] = R1; ap++
  ...
  [ap] = Rn; ap++
  -- store the pointer to allocated constructor data
  [ap] = [ap - n - 3] + 2; ap++
```

Another possiblity would be to use Cairo allocation hints, but this would actually result in more extra memory being used up for constructor data with more than 3 arguments.

Case switches. The JuvixReg `case` instruction is compiled to a relative jump into a jump table for case branches. In this way, any case switch can be executed in two jumps, regardless of the number of branches. More concretely,

```
case r {
  c1: goto B1
  ...
  cn: goto Bn
}
```

is compiled to

```
  jmp rel [R]
  jmp B1
  ...
  jmp Bn
```

```
B1: ...
...
Bn: ...
```

The reference $[R]$ fetches the CID of the constructor data. We use the CID as the relative offset to jump into the jump table for case branches. The form of the CID discussed in the previous paragraph ($2i + 1$ for constructor number i) guarantees that we jump to the ith jump instruction.

Closures. Memory layout for closures is:

- function id (FUID),
- $9 - s$ where s is the number of arguments stored in the closure,
- $9 - k$ where k is the number of arguments still expected by the function,
- s memory cells for the stored arguments.

The total number of arguments for the function is equal to $s + k$. The maximum number of arguments a function can have is 8. We store $9 - s$ and $9 - k$ instead of s and k to make it easier to implement closure calls and extensions. The FUID is an offset into a global call table used by the function `juvix_call_closure` implementing closure calls. Closure extension is implemented by the function `juvix_extend_closure`. The CASM code for these functions can be found in Appendix C.

Peephole optimization. Basic peephole optimization [ALSU07, Section 8.7] is run on the generated CASM code, which replaces certain instruction sequences with equivalent more efficient ones. For example, the sequence

```
call rel 3
ret
jmp L
```

is replaced with

```
call L
ret
```

The sequence

```
jmp L
L:
```

is replaced with

```
L:
```

The sequence

```
call rel 3
ret
[ap] = [fp - 4]; ap++
[ap] = [fp - 3]; ap++
ret
```

is replaced with

```
ret
```

And so on.

4.6. Cairo bytecode

Cairo bytecode [GPR21, Section 4.4] is a binary representation of Cairo assembly. Each CASM instruction takes up one or two Cairo memory words (field elements). Generally, the second word is needed when the instruction uses a non-offset immediate constant, e.g., in [ap] = [ap − 1] + 3; ap++ (the constant 3) or in `call L` (the label constant `L`).

Our internal representation of CASM is translated directly to Cairo bytecode. The extra instructions not present in actual CASM are first compiled to appropriate CASM instructions. Suitable initialization and finalization Cairo bytecode is added. The result is saved in a `*.json` file that can be read and executed by a special Juvix wrapper for the Rust Cairo VM [Lam24] (see Appendix A). The format of the generated file is compatible with the standard Cairo VM, but some of the hints used are implemented only in the Juvix Cairo VM wrapper.

5. Conclusion

This report describes the backbone of the Juvix backend compilation pipeline and the specific issues related to the generation of Cairo assembly. The read-only memory model of Cairo fits well with the purely functional nature of Juvix. The allocated constructor data is never modified. Neither are the closures – closure extension allocates a new closure. The only destructive assigments are made to local temporary variables in JuvixReg. By converting the code to SSA, we can make each variable be assigned only once, which then translates more easily to Cairo memory manipulation.

A major difficulty is caused by the requirement for continuous memory access in Cairo. To accommodate it, we divide JuvixReg code into basic blocks and associate a fixed `fp`-offset to each temporary variable assigned in the block. A transition to the next block, when the next `fp`-offset can no longer be statically determined, is done via a relative call with live variables transferred as arguments. The call "resets" the next `fp`-offset back to `0`.

The earlier parts of the pipeline progressively compile down the high-level functional features of Juvix:

- pattern matching compilation in JuvixCore breaks down complex pattern matches into one-level case-expressions,
- lambda-lifting on JuvixCore removes anonymous and local functions,
- type erasure in JuvixCore removes runtime type information overhead from polymorphic functions,
- application translation selects the right JuvixTree operation for each JuvixCore application (direct call, static or dynamic closure call, closure allocation or extension, constructor data allocation),
- dynamic closure call compilation in JuvixTree generates efficient code for call sites with possible partial application or overapplication,
- translation to JuvixAsm linearizes JuvixTree expressions into sequences of stack-based instructions and introduces the distinction between tail and non-tail instructions,
- translation to JuvixReg generates three-address code amenable to further transformation and analysis with classical compiler theory techniques.

All these transformations are inspired by or directly implement established methods from the (functional) compiler construction literature.

Acknowledgements

The author thanks the entire Juvix team, including Jonathan Prieto-Cubides, Jan Mas Rovira and Paul Cadman. The overall design and most of the implementation of the parts of the Juvix backend pipeline detailed in this report were done by the author. However, the input from the rest of the Juvix team helped to refine and debug the Cairo backend pipeline, especially the components related to JuvixCore. Jan Mas Rovira implemented eta-expansion and lambda-lifting, which turned out to be more tricky than anticipated.

The author also thanks Xuyang Song and Carlo Modica for feedback on the Cairo VM from a cryptographer's perspective and for clarifications on Cairo features needed for shielded Anoma applications.

References

ALSU07. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2nd edition, 2007. (cit. on pp. 26, 33, 34, and 40.)

App98. Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998. (cit. on p. 30.)

BG23. Jeremy Bruestle and Paul Gafni. RISC Zero zkVM: scalable, transparent arguments of RISC-V integrity. Draft, 2023. (cit. on p. 3.)

CT23. Keith D. Cooper and Linda Torczon. *Engineering a Compiler.* Morgan Kaufmann, 3rd edition, 2023. (cit. on pp. 25, 26, 30, and 31.)

Cza23. Lukasz Czajka. The Core language of Juvix. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8268849, `doi:10.5281/zenodo.8268850`. (cit. on pp. 3 and 12.)

DS04. Olivier Danvy and Ulrik Schultz. Lambda-lifting in quadratic time. *J. Funct. Log. Program.*, 2004. (cit. on p. 14.)

GPR21. Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo - a Turing-complete STARK-friendly CPU architecture. *IACR Cryptol. ePrint Arch.*, 2021. URL: https://eprint.iacr.org/2021/1063. (cit. on pp. 2, 4, 9, 38, and 41.)

GYB23. Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: a unified architecture for full-stack decentralised applications. *Anoma Research Topics*, Aug 2023. URL: https://doi.org/10.5281/zenodo.8279841, `doi:10.5281/zenodo.8279842`. (cit. on p. 2.)

Hel24. Heliax AG. Juvix Compiler, 2024. URL: https://github.com/anoma/juvix/. (cit. on p. 2.)

JL92. Simon Peyton Jones and David Lester. *Implementing functional languages: a tutorial.* Prentice Hall, 1992. (cit. on p. 14.)

Jon87. Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987. (cit. on p. 14.)

Lam24. LambdaClass. Cairo VM in Rust, 2024. URL: https://github.com/lambdaclass/cairo-vm. (cit. on pp. 9 and 41.)

Ler05. Xavier Leroy. From Krivine's machine to the Caml implementations. Invited talk at the KAZAM workshop, 2005. URL: https://xavierleroy.org/talks/zam-kazam05.pdf. (cit. on p. 22.)

Mar08. Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML*, pages 35–46, 2008. (cit. on p. 14.)

MJ06. Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006. (cit. on p. 22.)

A. Juvix CLI

In Juvix version `0.6.*`, to compile a Juvix program to Cairo bytecode type

```
juvix compile cairo Program.juvix
```

This produces a `Program.json` file. To execute it, you need the Juvix wrapper for the Cairo VM available at https://github.com/anoma/juvix-cairo-vm.

It is also possible to compile Juvix to Cairo Assembly (CASM), or in fact to any of the IRs mentioned in Section 4. Type:

```
juvix dev compile casm Program.juvix
```

This creates a file `Program.casm`. You can edit this file, and execute it in the Juvix CASM interpreter with:

```
juvix dev casm run Program.casm
```

CASM program examples appearing in this document can all be run in the Juvix CASM interpreter.

To see all commands related to CASM, type:

```
juvix dev casm --help
```

To see all available IR targets for the `dev compile` command, type:

```
juvix dev compile --help
```

Similarly to CASM, there is an interpreter or evaluator for each IR. To see commands related to an IR xxx (`core`, `tree`, `asm`, `reg`) type:

```
juvix dev xxx --help
```

B. Example program in different IRs

This appendix presents the respresentations of the `partition` function in all of the IRs described in the report. The `partition` function partitions a list into two lists: one containing the elements satisfying a given predicate `f`, the other containing the elements not satisfying `f`.

B.1. Juvix

```
type Pair A B := pair : A -> B -> Pair A B;

type List A :=
  | nil : List A
  | cons : A -> List A -> List A;

partition {A} (f : A → Bool) : List A → Pair (List A) (List A)
  | nil := pair nil nil
  | (cons x xs) :=
    case partition f xs of
      pair l1 l2 :=
        if
          | f x := pair (cons x l1) l2
          | else := pair l1 (cons x l2);
```

B.2. JuvixCore

```
type Pair {
  pair : Pi A : Type, Pi B : Type, A -> B -> Pair A B;
};
type List {
  nil : Pi A : Type, List A;
  cons : Pi A : Type, A -> List A -> List A;
};
def partition
  : Pi A : Type, (A -> Bool) -> List A -> Pair (List A) (List A) :=
  \(A : Type) \(f : A -> Bool) \(_X : List A)
    case _X of {
      cons (_X' : Type) (x : A) (xs : List A) :=
        case partition A f xs of {
          pair _ _ (l1 : List A) (l2 : List A) :=
            if f x then
              pair (List A) (List A) (cons A x l1) l2
            else
              pair (List A) (List A) l1 (cons A x l2)
        };
      nil _ := pair (List A) (List A) (nil A) (nil A)
    };
```

B.3. Stripped JuvixCore

```
type Pair {
  pair : Any -> Any -> Pair Any Any;
};
type List {
  nil : List Any;
  cons : Any -> List Any -> List Any;
};
def partition
  : (Any -> Bool) -> List Any -> Pair (List Any) (List Any) :=
  \(f : Any → Bool) \(_X : List Any)
    case _X of {
      cons x (xs : List Any) :=
        case partition f xs of {
          pair (l1 : List Any) (l2 : List Any) :=
            if f x then
              pair (cons x l1) l2
            else
              pair l1 (cons x l2)
        };
      nil := pair nil nil
    };
```

B.4. JuvixTree

```
type Pair {
  pair : (*, *) -> Pair;
}
type List {
  nil : List;
  cons : (*, List) -> List;
}
function partition(* -> bool, List) : Pair {
  case[List](arg[1]) {
    cons: save {
      case[Pair](call[partition](arg[0], tmp[0].cons[1])) {
        pair: save {
          br(ccall(arg[0], tmp[0].cons[0])) {
            true:
              alloc[pair](
```

```
              alloc[cons](tmp[0].cons[0], tmp[1].pair[0]),
              tmp[1].pair[1]
            )
          false:
            alloc[pair](
              tmp[1].pair[0],
              alloc[cons](tmp[0].cons[0], tmp[1].pair[1])
            )
        }
      }
    }
  }
  nil: alloc[pair](alloc[nil](), alloc[nil]())
 }
}
```

B.5. JuvixAsm

```
function partition(* -> bool, List) : Pair {
  push arg[1];
  case List {
    cons: {
      tsave {
        push tmp[0].cons[1];
        push arg[0];
        call partition;
        case Pair {
          pair: {
            tsave {
              push tmp[0].cons[0];
              push arg[0];
              call apply_1;
              br {
                true: {
                  push tmp[1].pair[1];
                  push tmp[1].pair[0];
                  push tmp[0].cons[0];
                  alloc cons;
                  alloc pair;
                  ret;
```

```
              };
              false: {
                push tmp[1].pair[1];
                push tmp[0].cons[0];
                alloc cons;
                push tmp[1].pair[0];
                alloc pair;
                ret;
              };
            };
          };
        };
      };
    };
  };
  nil: {
    pop;
    alloc nil;
    alloc nil;
    alloc pair;
    ret;
  };
};
}
```

B.6. JuvixReg

```
function partition(* -> bool, List) : Pair {
  tmp[2] = arg[1];
  case[List] tmp[2] {
    cons: {
      tmp[0] = tmp[2];
      tmp[2] = tmp[0].cons[1];
      tmp[3] = arg[0];
      tmp[2] = call partition (tmp[3], tmp[2]);
      tmp[1] = tmp[2];
      tmp[2] = tmp[0].cons[0];
      tmp[3] = arg[0];
      tmp[2] = call apply_1 (tmp[3], tmp[2]);
      br tmp[2] {
```

```
        true: {
          tmp[2] = tmp[1].pair[1];
          tmp[3] = tmp[1].pair[0];
          tmp[4] = tmp[0].cons[0];
          tmp[3] = alloc cons (tmp[4], tmp[3]);
          tmp[2] = alloc pair (tmp[3], tmp[2]);
          ret tmp[2];
        };
        false: {
          tmp[2] = tmp[1].pair[1];
          tmp[3] = tmp[0].cons[0];
          tmp[2] = alloc cons (tmp[3], tmp[2]);
          tmp[3] = tmp[1].pair[0];
          tmp[2] = alloc pair (tmp[3], tmp[2]);
          ret tmp[2];
        };
      };
    };
    nil: {
      tmp[2] = alloc nil ();
      tmp[3] = alloc nil ();
      tmp[2] = alloc pair (tmp[3], tmp[2]);
      ret tmp[2];
    };
  };
}
```

B.7. JuvixReg in SSA

```
function partition(* → bool, List) : Pair {
  tmp[0] = arg[1];
  case[List] tmp[0] {
    cons: {
      tmp[1] = tmp[0];
      tmp[2] = tmp[1].cons[1];
      tmp[3] = arg[0];
      tmp[4] = call partition (tmp[3], tmp[2]);
      tmp[5] = tmp[4];
      tmp[6] = tmp[1].cons[0];
      tmp[7] = arg[0];
```

```
      tmp[8] = call apply_1 (tmp[7], tmp[6]);
      br tmp[8] {
        true: {
          tmp[9] = tmp[5].pair[1];
          tmp[10] = tmp[5].pair[0];
          tmp[11] = tmp[1].cons[0];
          tmp[12] = alloc cons (tmp[11], tmp[10]);
          tmp[13] = alloc pair (tmp[12], tmp[9]);
          ret tmp[13];
        };
        false: {
          tmp[9] = tmp[5].pair[1];
          tmp[10] = tmp[1].cons[0];
          tmp[11] = alloc cons (tmp[10], tmp[9]);
          tmp[12] = tmp[5].pair[0];
          tmp[13] = alloc pair (tmp[12], tmp[11]);
          ret tmp[13];
        };
      };
    };
    nil: {
      tmp[1] = alloc nil ();
      tmp[2] = alloc nil ();
      tmp[3] = alloc pair (tmp[2], tmp[1]);
      ret tmp[3];
    };
  };
}
```

B.8. Optimized JuvixReg

```
function partition(* → bool, List) : Pair {
  case[List] arg[1] {
    cons: {
      tmp[0] = arg[1].cons[1];
      tmp[1] = call partition (arg[0], tmp[0]);
      tmp[2] = arg[1].cons[0];
      tmp[3] = call apply_1 (arg[0], tmp[2]);
      br tmp[3] {
        true: {
```

```
          tmp[4] = tmp[1].pair[1];
          tmp[5] = tmp[1].pair[0];
          tmp[6] = arg[1].cons[0];
          tmp[7] = alloc cons (tmp[6], tmp[5]);
          tmp[8] = alloc pair (tmp[7], tmp[4]);
          ret tmp[8];
        };
        false: {
          tmp[4] = tmp[1].pair[1];
          tmp[5] = arg[1].cons[0];
          tmp[6] = alloc cons (tmp[5], tmp[4]);
          tmp[7] = tmp[1].pair[0];
          tmp[8] = alloc pair (tmp[7], tmp[6]);
          ret tmp[8];
        };
      };
    };
    nil: {
      tmp[0] = alloc nil ();
      tmp[1] = alloc nil ();
      tmp[2] = alloc pair (tmp[1], tmp[0]);
      ret tmp[2];
    };
  };
}
```

B.9. JuvixReg basic blocks

```
block A, live: (arg[0], arg[1]) {
  case[List] arg[1] {
    cons: { goto B; };
    nil: { goto C; };
  };
}

block B, live: (arg[0], arg[1]) {
  tmp[0] = arg[1].cons[1];
  tmp[1] = call partition (arg[0], tmp[0]);
  goto B1;
}
```

```
block B1, live: (arg[0], arg[1], tmp[1]) {
  tmp[2] = arg[1].cons[0];
  tmp[3] = call apply_1 (arg[0], tmp[2]);
  goto B2;
}

block B2, live: (tmp[3], tmp[1], arg[1]) {
  br tmp[3] {
    true: { goto B3; };
    false: { goto B4; };
  };
}

block B3, live: (tmp[1], arg[1]) {
  tmp[4] = tmp[1].pair[1];
  tmp[5] = tmp[1].pair[0];
  tmp[6] = arg[1].cons[0];
  tmp[7] = alloc cons (tmp[6], tmp[5]);
  tmp[8] = alloc pair (tmp[7], tmp[4]);
  ret tmp[8];
}

block B4, live: (tmp[1], arg[1]) {
  tmp[4] = tmp[1].pair[1];
  tmp[5] = arg[1].cons[0];
  tmp[6] = alloc cons (tmp[5], tmp[4]);
  tmp[7] = tmp[1].pair[0];
  tmp[8] = alloc pair (tmp[7], tmp[6]);
  ret tmp[8];
}

block C, live: () {
  tmp[0] = alloc nil ();
  tmp[1] = alloc nil ();
  tmp[2] = alloc pair (tmp[1], tmp[0]);
  ret tmp[2];
}
```

B.10. CASM

```
partition:
  jmp rel [[fp - 4]]
  jmp label_17
  jmp label_18
label_17:
  -- block C
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 1; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 1; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = [fp + 10]; ap++
  [ap] = [fp + 4]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp + 16]; ap++
  ret
label_18:
  -- block B
  [ap] = [[fp - 4] + 2]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 3]; ap++
  call partition
  [ap] = [fp - 3]; ap++
  [ap] = [fp - 4]; ap++
  call rel 3
  ret
  -- block B1
  [ap] = [[fp - 3] + 1]; ap++
  [ap] = [fp - 6]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 4]; ap++
  call apply_1
  [ap] = [fp - 3]; ap++
```

```
  [ap] = [fp - 5]; ap++
  call rel 3
  ret
  -- block B2
  jmp label_20 if [fp - 5] != 0
  -- block B3
  [ap] = [[fp - 3] + 1]; ap++
  [ap] = [[fp - 3]]; ap++
  [ap] = [[fp - 4] + 1]; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 3; ap++
  [ap] = [fp + 2]; ap++
  [ap] = [fp + 1]; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = [fp + 7]; ap++
  [ap] = [fp]; ap++
  [ap] = [fp - 6]; ap++
  [ap] = [fp + 15]; ap++
  ret
label_20:
  -- block B4
  [ap] = [[fp - 3] + 1]; ap++
  [ap] = [[fp - 4] + 1]; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = 3; ap++
  [ap] = [fp + 1]; ap++
  [ap] = [fp]; ap++
  [ap] = [[fp - 3]]; ap++
  call juvix_get_regs
  [ap] = [ap - 2] + 3; ap++
  [ap] = [fp + 10]; ap++
  [ap] = [fp + 6]; ap++
  [ap] = [fp - 6]; ap++
  [ap] = [fp + 15]; ap++
  ret
```

C. CASM runtime

In this section, we list the CASM code of the Juvix runtime functions dealing with closure calls and closure extensions.

C.1. Closure call

```
-- [fp - 3]: closure;
-- [fp - 4 - k]: argument k to closure call (0-based)
-- [fp - 4 - n]: builtin pointer, where n = number of supplied args
juvix_call_closure:
  -- jmp rel (9 - argsnum)
  jmp rel [[fp - 3] + 2]
  -- copy extra args: builtin ptr + args
  [ap] = [fp - 12]; ap++
  [ap] = [fp - 11]; ap++
  [ap] = [fp - 10]; ap++
  [ap] = [fp - 9]; ap++
  [ap] = [fp - 8]; ap++
  [ap] = [fp - 7]; ap++
  [ap] = [fp - 6]; ap++
  [ap] = [fp - 5]; ap++
  [ap] = [fp - 4]; ap++
  -- copy stored args: jmp rel (9 - sargs)
  jmp rel [[fp - 3] + 1]
  [ap] = [[fp - 3] + 10]; ap++
  [ap] = [[fp - 3] + 9]; ap++
  [ap] = [[fp - 3] + 8]; ap++
  [ap] = [[fp - 3] + 7]; ap++
  [ap] = [[fp - 3] + 6]; ap++
  [ap] = [[fp - 3] + 5]; ap++
  [ap] = [[fp - 3] + 4]; ap++
  [ap] = [[fp - 3] + 3]; ap++
  -- call the closure function
  jmp rel [[fp - 3]]
-- Here there is a global function call table specific
-- to a given program.
```

C.2. Closure extension

```
-- [fp - 3]: closure
```

```
-- [fp - 4]: n = the number of arguments to extend with
-- [fp - 4 - k]: argument n - k - 1 (reverse order!) (k is 0-based)
-- On return:
-- [ap - 1]: new closure
-- This procedure doesn't accept or return the builtins pointer.
juvix_extend_closure:
  -- copy stored args reversing them;
  -- to copy the stored args to the new closure
  -- we need to jump forward, so the stored args
  -- need to be available at consecutive memory
  -- addresses backwards
  jmp rel [[fp - 3] + 1]
  [ap] = [[fp - 3] + 10]; ap++
  [ap] = [[fp - 3] + 9]; ap++
  [ap] = [[fp - 3] + 8]; ap++
  [ap] = [[fp - 3] + 7]; ap++
  [ap] = [[fp - 3] + 6]; ap++
  [ap] = [[fp - 3] + 5]; ap++
  [ap] = [[fp - 3] + 4]; ap++
  [ap] = [[fp - 3] + 3]; ap++
  -- the following ensures continuous memory use
  -- with a compile-time constant offset for local
  -- variables
  [ap] = 10; ap++
  [ap] = [[fp - 3] + 1]; ap++
  [ap] = [ap - 2] - [ap - 1]; ap++
  jmp rel [ap - 1]
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  [ap] = [ap - 1]; ap++
  -- now ap = fp + 11
  -- alloc
  call juvix_get_regs
```

```
-- now ap = fp + 15
-- [fp + 15] = pointer to new closure
[ap] = [ap - 2] + 8; ap++
-- [fp + 16] = 9 - sargs
[ap] = [[fp - 3] + 1]; ap++
-- [fp + 17] = 9 - argsnum (expected)
[ap] = [[fp - 3] + 2]; ap++
-- [fp + 18] = 9
[ap] = 9; ap++
-- [fp + 19] = sargs
[ap] = [fp + 18] - [fp + 16]; ap++
-- [fp + 20] = 9 - n
[ap] = [fp + 18] - [fp - 4]; ap++
-- closure header
[ap] = [[fp - 3]]; ap++
[ap] = [fp + 16] - [fp - 4]; ap++
[ap] = [fp + 17] + [fp - 4]; ap++
-- copy stored args: jmp rel (9 - sargs)
jmp rel [fp + 16]
[ap] = [fp + 7]; ap++
[ap] = [fp + 6]; ap++
[ap] = [fp + 5]; ap++
[ap] = [fp + 4]; ap++
[ap] = [fp + 3]; ap++
[ap] = [fp + 2]; ap++
[ap] = [fp + 1]; ap++
[ap] = [fp]; ap++
-- copy extra args: jmp rel (9 - extra args num)
jmp rel [fp + 20]
[ap] = [fp - 12]; ap++
[ap] = [fp - 11]; ap++
[ap] = [fp - 10]; ap++
[ap] = [fp - 9]; ap++
[ap] = [fp - 8]; ap++
[ap] = [fp - 7]; ap++
[ap] = [fp - 6]; ap++
[ap] = [fp - 5]; ap++
-- return value
```

```
[ap] = [fp + 15]; ap++
ret
```