1 The Factor Compiler

Though there are projects for this

If we could sort programming languages by the fuzzy notions we tend to have about how "high-level" they are, toward the high end we'd find dynamically-typed languages like Python, Ruby, and PHP—all of which are generally more interpreted than compiled. Despite being as high-level as these popular languages, Factor's implementation is driven by performance. Factor source is always compiled to native machine code using either its simple, non-optimizing compiler or (more typically) the optimizing compiler that performs several sorts of data and control flow analyses. In this section, we look at the general architecture of Factor's implementation, after which we place a particular emphasis on the transformations performed by the optimizing compiler.

1.1 High-level Optimizations

To manipulate source code abstractly, we must have at least one intermediate representation (IR)—a data structure representing the instructions. It's common to convert between several IRs during compilation, as each form offers different properties that facilitate particular analyses. The Factor compiler optimizes code in passes across two different IRs: first at a high-level using the `compiler.tree` vocabulary, then at a low-level with the `compiler.cfg` vocabulary.

The high-level IR arranges code into a vector of `node` objects, which may themselves have children consisting of vectors of node—a tree structure that lends to the name `compiler.tree`. This ordered sequence of nodes represents control flow in a way that's effectively simple, annotated stack code. Listing 1 on the following page shows the definitions of the tuples that represent the "instruction set" of this stack code. Each object inherits (directly or indirectly) from the `node` class, which itself inherits from **`identity-tuple`**. This is a tuple whose **`equal?`** method is defined to always return **`f`** so that no two instances are equivalent unless they are the same instance.

Notice that most nodes define some sort of `in-d` and `out-d` slots, which mark each of them with the input and output data stacks. This represents the flow of data through the program. Here, stack values are denoted simply by integers, giving each value a unique identifier. An `#introduce` instance is inserted wherever the next node requires stack values that have not yet been named. Thus, while `#introduce` has no `in-d`, its `out-d` introduces the necessary stack values. Similarly, `#return` is inserted at the end of the sequence to indicate the final state of the data stack with its `in-d` slot.

The most basic operations of a stack language are, of course, pushing literals and calling functions that pop inputs and push outputs. The `#push` node thus has a `literal` slot and an `out-d` slot, giving a name to the single element it pushes to the data stack. `#call`, of course, is used for normal word invocations. The `in-d` and `out-d` slots effectively serve as the stack effect declaration. In later analyses, data about the word's definition may be stored across the `body`, `method`, `class`, and `info` slots.

The word `build-tree` takes a Factor quotation and constructs the equivalent high-level IR form. In Listing 2 on page 3, we see the output of the simple example `[ 1 + ] build-tree`. Note that `T{ class { slot1 value1 } { slot2 value2 } ... }` is the syntax for tuple literals. The first node is a `#push` for the 1 literal. Since `+` needs two input values, an `#introduce` pushes a new "phantom" value. `+` gets turned into a `#call` instance. Notice the `in-d` slot refers to the values in

```
TUPLE: node < identity-tuple ;

TUPLE: #introduce < node out-d ;
TUPLE: #return < node in-d info ;

TUPLE: #push < node literal out-d ;
TUPLE: #call < node word in-d out-d body method class info ;

TUPLE: #renaming < node ;
TUPLE: #copy < #renaming in-d out-d ;
TUPLE: #shuffle < #renaming mapping in-d out-d in-r out-r ;

TUPLE: #declare < node declaration ;

TUPLE: #terminate < node in-d in-r ;

TUPLE: #branch < node in-d children live-branches ;
TUPLE: #if < #branch ;
TUPLE: #dispatch < #branch ;

TUPLE: #phi < node phi-in-d phi-info-d out-d terminated ;

TUPLE: #recursive < node in-d word label loop? child ;
TUPLE: #enter-recursive < node in-d out-d label info ;
TUPLE: #call-recursive < node label in-d out-d info ;
TUPLE: #return-recursive < #renaming in-d out-d label info ;

TUPLE: #alien-node < node params ;
TUPLE: #alien-invoke < #alien-node in-d out-d ;
TUPLE: #alien-indirect < #alien-node in-d out-d ;
TUPLE: #alien-assembly < #alien-node in-d out-d ;
TUPLE: #alien-callback < node params child ;
```

Listing 1: High-level IR nodes

```
V{
    T{ #push { literal 1 } { out-d { 6256273 } } }
    T{ #introduce { out-d { 6256274 } } }
    T{ #call
        { word + }
        { in-d V{ 6256274 6256273 } }
        { out-d { 6256275 } }
    }
    T{ #return { in-d V{ 6256275 } } }
}
```

Listing 2: `[ 1 + ] build-tree`

the order that they're passed to the word, not necessarily the order they've been introduced in the IR. The sum is pushed to the data stack, so the `out-d` slot is a singleton that names this value. Finally, `#return` indicates the end of the routine, its `in-d` containing the value left on the stack (the sum pushed by `#call`).

```
V{
    T{ #introduce { out-d { 6256132 6256133 } } }
    T{ #shuffle
        { mapping { { 6256134 6256133 } { 6256135 6256132 } } }
        { in-d V{ 6256132 6256133 } }
        { out-d V{ 6256134 6256135 } }
    }
    T{ #return { in-d V{ 6256134 6256135 } } }
}
```

Listing 3: `[ swap ] build-tree`

The next tuples in Listing 1 on the preceding page reassign existing values on the stack to fresh identifiers. The `#renaming` superclass has the two subclasses `#copy` and `#shuffle`. The former represents the bijection from elements of `in-d` to elements of `out-d` in the same position; corresponding values are copies of each other. The latter represents a more general mapping. Stack shufflers are translated to `#shuffle` nodes with `mapping` slots that dictate how the fresh values in `out-d` correspond to the input values in `in-d`. For instance, Listing 3 shows how **`swap`** takes in the values `6256132` and `6256133` and outputs `6256134` and `6256135`, where the former is mapped to the second element (`6256133`) and the latter to the first (`6256132`). Thus, `out-d` swaps the two elements of `in-d`, mapping them to fresh identifiers. The `in-r` and `out-r` slots of `#shuffle` correspond to the *retain* stack, which is an implementation detail beyond the scope of this discussion.

`#declare` is a miscellaneous node used for the `declare` primitive. It simply annotates type information to stack values, as in Listing 4 on the following page. `#terminate` is another one-

```
V{
    T{ #introduce { out-d { 6256069 } } }
    T{ #declare { declaration { { 6256069 fixnum } } } }
    T{ #return { in-d V{ 6256069 } } }
}
```

Listing 4: `[ { fixnum } declare ] build-tree`

```
V{
    T{ #push { literal "Error!" } { out-d { 6256051 } } }
    T{ #call
        { word throw }
        { in-d V{ 6256051 } }
        { out-d { } }
    }
    T{ #terminate { in-d V{ } } { in-r V{ } } }
    T{ #return { in-d V{ } } }
}
```

Listing 5: `[ "Error!" throw ] build-tree`

off node, but a much more interesting one. While Factor normally requires a balanced stack, sometimes we purposefully want to throw an error. `#terminate` is introduced where the program halts prematurely. When checking the stack height, it gets to be treated specially so that *terminated* stack effects unify with any other effect. That way, branches will still be balanced even if one of them unconditionally throws an error. Listing 5 shows `#terminate` being introduced by the **`throw`** word.

Next, Listing 1 on page 2 defines nodes for branching based off the superclass `#branch`. The `children` slot contains vectors of nodes representing different branches. `live-branches` is filled in during later analyses to indicate which branches are alive so that dead ones may be removed. For instance, `#if` will have two elements in its `children` slot representing the true and false branches. On the other hand, `#dispatch` has an arbitrary number of children. It corresponds to the `dispatch` primitive, which is an implementation detail of the generic word system used to speed up method dispatch.

You may have noted the emphasis on introducing new values in `out-d` slots. Even `#shuffle`s output fresh identifiers, letting their values be determined by its `mapping`. The reason for this is that `compiler.tree` uses static single assignment (SSA) form, wherein every variable is defined by exactly one statement. This simplifies the properties of variables, which helps optimizations perform faster and with better results. By giving unique names to the targets of each assignment, the SSA property is guaranteed. However, `#branch`es introduce ambiguity: after, say, an `#if`, what will the identifiers in `out-d` be? It depends on which branch is taken. To remedy this problem, after any `#branch` node, Factor will place a `#phi` node—the classical SSA "phony function". The `phi-in-d` slot seen in Listing 1 on page 2 is a sequence of sequences; each one corresponds to the `out-d` of the child at the same position in the `children` of the preceding node. The `#phi`'s `out-d` gives unique names to the output values, thus ensuring the SSA property. Though it doesn't perform any literal computation, conceptually it select the "correct" `out-d` depending on the control flow.

```
V{
    T{ #introduce { out-d { 6256247 } } }
    T{ #if
        { in-d { 6256247 } }
        { children
            {
                V{
                    T{ #push
                        { literal 1 }
                        { out-d { 6256248 } }
                    }
                }
                V{
                    T{ #push
                        { literal 2 }
                        { out-d { 6256249 } }
                    }
                }
            }
        }
    }
    T{ #phi
        { phi-in-d { { 6256248 } { 6256249 } } }
        { out-d { 6256250 } }
        { terminated V{ f f } }
    }
    T{ #return { in-d V{ 6256250 } } }
}
```

Listing 6: `[ [ 1 ] [ 2 ] if ] build-tree`

For example, the `#phi` in Listing 6 will select between the `6256248` return value of the first child or the `6256249` output of the second. Either way, we can refer to the result as `6256250` afterwards. The `terminated` slot of the `#phi` tells us if there was a `#terminate` in any of the branches.

The `#recursive` node encapsulates *inline recursive* words. In Factor, words may be annotated with simple compiler declarations, which guide optimizations. If we follow a standard colon definition with the **`inline`** word, we're saying that its definition can be spliced into the call-site, rather than generating code to jump to a subroutine. Inline words that call themselves must additionally be declared **`recursive`**. For example, we could write `: foo ( -- ) foo ; inline recursive`. The nodes `#enter-recursive`, `#call-recursive`, and `#return-recursive` denote different stages of the recursion—the beginning, recursive call, and end, respectively. They carry around a lot of metadata about the nature of the recursion, but it doesn't serve our purposes to get into the details. Similarly, we gloss over the final nodes of Listing 1 on page 2 correspond to Factor's foreign function interface (FFI) vocabulary, called `alien`. At a high level, these are used to make calls to C libraries from within Factor.