ManyWorlds: Combinatorial Programming with Functions

Jo Devriendt
Nonfiction Software, Belgium https://nonfictionsoftware.com

Abstract

The ManyWorlds programming language provides an abstract, high-level syntax built on a small set of core concepts to specify combinatorial problems. It uses total functions as fundamental building blocks and draws inspiration from the higher-order functions fold, map and filter to compactly aggregate expressions. ManyWorlds follows the knowledge base paradigm by stressing that each specification represents a set of many possible 'worlds' that a user can interact with in different ways. Finding an (optimal) world is complemented by counting the number of worlds, calculating the intersection of all worlds, and explaining why a world does – or does not – exist.

ManyWorlds values accessibility and ease of development. It provides an online IDE, helpful parsing error messages, line-based explanations of inconsistency and expression evaluation support. The ManyWorlds compiler translates a high-level specification to an integer program, where compactness of the compiled problem is paramount. This compiler is still under development, but it already supports much of the envisioned language and is ready for third-party experimentation.

This paper gives a high-level introduction to ManyWorlds and compares it to MiniZinc on the On-Call Rostering problem from previous MiniZinc challenges.

2012 ACM Subject Classification Computing methodologies → Logic programming and answer set programming

Keywords and phrases Constraint programming, Functional programming, First-order logic, Knowledge base paradigm, Development support, Combinatorial optimization

1 Introduction

In its most abstract form, *combinatorial programming* consists of writing down unambiguous formulas that a computer can interpret and solve. Those formulas (or expressions) range from simple propositional clauses (SAT solving) over mathematical (in)equalities (pseudo-Boolean solving, integer programming, mathematical programming, . . .) to logic-based formalisms (first order logic and extensions, answer set programming, satisfiability modulo theories, . . .), perhaps with special purpose constraints (constraint programming).

From a combinatorics point of view, all these formalisms allow to describe a set of *possible worlds* (solutions) by an unambiguous expression. The backend solvers, grounders, compilers and interpreters for these formalisms then reason about this set of possible worlds. The focus of a formalism or a backend may differ. E.g., it may specialize in finding an optimal world (optimization), on deciding whether at least one world exists (satisfiability), on counting the number of worlds (counting), on finding the intersection of all worlds (propagation), etc. But in the end, it all boils down to reasoning about a set of possible worlds.

ManyWorlds is a new high-level combinatorial programming language that aims to be as accessible as possible. Its syntax and semantics are centered squarely on the concept of a function, which both programmers and high-schoolers are familiar with. This focus on functions keeps the syntax and semantics simple yet expressive: expressions are nested function applications, constraints are expressions that must be true, and possible worlds (solutions) are instantiations of functions that satisfy the constraints. To further improve accessibility, ManyWorlds features extensive debugging support, clear error reporting, low-mathematics syntax, arbitrary precision integer arithmetic, shorthand syntactic sugar, and in general tries to offload tedium to the compiler.

One of ManyWorlds' goals is to support the *Knowledge Base Paradigm* [18] where a program closely mirrors a problem domain description, and a single program can be used for multiple computational tasks.

Beyond its syntax and semantics, ManyWorlds aims to provide a beginner-friendly user experience. A no-install online editor with basic syntax highlighting and in-browser syntax checking,[1] a precompiled docker image,[2] and source code with an open source license [3] are publically available. Every feature described in this paper, unless explicitly noted, is fully implemented, tested, and ready for third-party experimentation.

2 Syntax and semantics

2.1 Functional core

Values in ManyWorlds can be one of three primitive types: `bool`, `int` and `string` – representing, respectively, the Boolean values `true`, `false`; integer numbers `-1`, `0`, `1`, `2` etc.; and character string values `"hello world"`, etc. Typing is strict: only explicit conversion is possible.

From this, type signatures can be built, e.g., `int, string -> bool` is a signature representing pairs of integers and strings mapped to true or false. ManyWorlds has the following infix and prefix **builtin operators** with associated signatures:

- `+`, `-`, `*`: `int, int -> int`;
- `-` (unary minus): `int -> int`;
- `not`: `bool -> bool`;
- `and`, `or`, `xor`, `implies` (material implication): `bool, bool -> bool`;
- `>`, `<`, `>=`, `<=`: `int, int -> bool`;
- `=`, `!=`: overloaded for `int, int -> bool` and `string, string -> bool` and `bool, bool -> bool`;
- `... if ... else ...` (Python-style ternary conditional): overloaded for `string, bool, string -> string` and `int, bool, int -> int` and `bool, bool, bool -> bool`.

ManyWorlds also sports a number of **builtin functions**:

- `abs` (absolute value): `int -> int`;
- `div`, `rem` (truncated integer division and associated remainder): `int, int -> int`;
- `min`, `max`: `int* -> int`;
- `count`: `bool* -> int`;
- `same`, `distinct` (whether all arguments are equal or different): overloaded for `int* -> bool` and `string* -> bool`.

`<type>*` denotes that the builtin function is overloaded for any number of arguments (including zero). E.g., both `count(p(),q(),r())` and `count(p())` are valid expressions.

Any function in ManyWorlds, including operators and builtin functions, is **total** over its domain. This means that typically undefined operations (e.g., division by zero) do yield a valid result. For builtin functions and operators, ManyWorlds simply defines this result: division by zero yields zero, remainder by zero yields zero, and `min` and `max` applied to zero arguments yield zero. Enforcing totality is often employed by theorem provers to keep

[1] `https://manyworlds.site`
[2] `https://hub.docker.com/r/nonfictionsoftware/manyworlds`
[3] `https://gitlab.com/nonfiction-software/manyworlds`

the semantics simple. E.g., Microsoft's Z3 [6] allows division by zero, and ManyWorlds has adopted Coq's convention that division and remainder by zero are zero [17].

A user can declare **user functions** with the same type signatures, except that the codomain of a user function must be finite. E.g., `declare f: int -> {0 .. 3}.` declares a user function `f` with signature `int -> int` whose codomain is the finite set $\{0, 1, 2, 3\}$. The reason that codomains must be finite is that this makes it simple to derive a finite range of possible values for any user function application expression. As a result, most expressions will have an easily deducible finite range. E.g., `f(x)+f(y)` can only take a finite set of values given that `f`'s codomain is finite. The reason that user function *domains* must *not* be similarly finite is that it is straightforward to derive the domain of a function based on the finite ranges of the arguments with which the function occurs in the program, so ManyWorlds does not require a user to provide these.

User functions are the beating heart of ManyWorlds. They represent constants (when given no input types), sets (when mapping a single input to `bool`), relations (when mapping multiple inputs to `bool`) or properties (typically mapping to `int` or `string`). A user function represents any function that matches its signature. E.g., the user function `declare f: int -> {0 .. 3}.` can represent any function from $\mathbb{Z}$ to $\{0, 1, 2, 3\}$. Hence, user functions fulfill the role of *decision variables* present in other combinatorial programming languages.

Expressions in ManyWorlds are constructed by composing values, builtin operators, builtin functions and user functions while adhering to natural typing and function application rules. E.g., `abs(f(1)) > f(0) implies f(f(10)) = 0` is a valid expression (given, e.g., a user function `declare f: int -> {0 .. 3}.`). To write a *constraint*, we just have to assert a Boolean expression at the top level of the program by ending it with a '`.`' (which is the end delimiter for all top-level expressions, including constraints and declarations).

A **candidate (world)** is a set of total functions that match the user function declarations in the program. Given a candidate, an expression evaluates to a value in the usual inductive manner: values evaluate to themselves, compound expressions take the evaluation of their subexpressions and map this to a value based on the corresponding total function (either from a builtin operator, from a builtin function, or from a user function with a matching total function in the candidate). A **(valid) world** or **solution** is a candidate that evaluates all constraints to `true`.

With all of this machinery, we can program the well-known Send More Money problem:

```
declare S, E, N, D, M, O, R, Y: -> {0 .. 9}.
M() > 0.
distinct(S(), E(), N(), D(), M(), O(), R(), Y()).
           1000*S() + 100*E() + 10*N() + D() +
           1000*M() + 100*O() + 10*R() + E() =
10000*M() + 1000*O() + 100*N() + 10*E() + Y() .
```

The first line declares eight constant user functions, while the following lines spell out the constraints that `M()` should be non-zero, all letters should take a distinct value, and the columnar addition should hold. This program allows one unique world which matches `S, E, N, D, M, O, R, Y` to the constant functions `9, 5, 6, 7, 1, 0, 8, 2`, respectively.

2.2 Enumeration definitions

To describe data, **enumeration definitions** uniquely fix a user function by enumerating input-output tuples for the function, and by passing a default output value for all of the non-enumerated tuples. E.g.,

```
declare item: string -> bool.
define item as {("i1",true), ("i2",true), ("i3",true)} default false.
```

The above fixes `item` to be the function mapping `{"i1", "i2", "i3"}` to `true` and all other strings to `false`. In other words, it represents the set `{"i1", "i2", "i3"}`.

Enumeration definitions allow user functions to represent input data, and hence, to also take on the role of *parameters* present in other combinatorial programming languages.

As it is cumbersome to write out these tuple enumerations by hand, ManyWorlds provides the possibility to call a Python function that generates a list of tuples. E.g., the below enumeration is equivalent to the above one:

```
declare item: string -> bool.
define item as
    {$python
    def item():
        return [("i"+str(k),True) for k in range(1,4)]
    $}
    default false.
```

2.3 Fold-Map-Filter

Many high-level expressions are *aggregations* of simpler expressions parametrized by some set of objects. For instance, in many combinatorial problems, the sum of the weights of some items is a central concept. The weight of a given item is a simple expression, and we could write it as a sum as follows:

```
weight("i1") + weight("i2") + weight("i3")
```

This is pretty cumbersome if we have hundreds of items, and if the set of items changes, we would prefer to not update the expression summing their weights.

For this, ManyWorlds uses the *fold-map-filter* (FMF) expression. It takes four arguments: a builtin *fold function*, a *map* expression, a *filter* expression, and a list of *scoped variables* shared by the map and filter expressions. The filter is a Boolean expression that is true for a finite amount of instantiations of the scoped variables. The map maps those variable instantiations to a list of expressions of an appropriate type for the fold function. The fold function combines the mapped expressions to a new expression. E.g., given the declarations and definition

```
declare item: string -> bool.
define item as {("i1",true), ("i2",true), ("i3",true)} default false.
declare weight: string -> {0 .. 5}.
declare value: string -> {0 .. 5}.
declare inKnapsack: string -> bool.
```

We can write the FMF expression

```
sum[ weight(x) for x where item(x) and inKnapsack(x) ]
```

Here, the fold function is `sum`, the map is `weight(x)`, the scoped variable is `x` and the filter is `item(x) and inKnapsack(x)`. Hopefully, it is intuitive that this expression represents the sum of the weights of all items in a knapsack, regardless of their amount. Using FMF expressions, we complete the knapsack example by adding a knapsack constraint and a maximization objective:

```
sum[ weight(x) for x where item(x) and inKnapsack(x) ] <= 10.
@maximize sum[ value(x) for x where item(x) and inKnapsack(x) ].
```

Let $\overline{x}$ be a tuple of scoped variables, $m(\overline{x})$ a map expression and $f(\overline{x})$ a filter expression. ManyWorlds supports the following **builtin fold functions**:

- `any/all/none`: whether $m(\overline{x})$ is true for any/all/no $\overline{x}$ where $f(\overline{x})$ holds;
- `sum/product`: the sum/product of all $m(\overline{x})$ for $\overline{x}$ where $f(\overline{x})$ holds;
- `min/max`: the minimum/maximum of all $m(\overline{x})$ for $\overline{x}$ where $f(\overline{x})$ holds;
- `distinct/same`: whether all $m(\overline{x})$ are distinct/equal for $\overline{x}$ where $f(\overline{x})$ holds;
- `count`: the number of $m(\overline{x})$ that are true for $\overline{x}$ where $f(\overline{x})$ holds;
- `even/odd`: whether the number of $f(\overline{x})$ that are true is even/odd for $\overline{x}$ where $f(\overline{x})$ holds.

FMF expressions are extremely flexible and the current list of builtin fold functions captures a lot of relevant combinatorial concepts. Note that even though FMF expressions are inspired by the fold, map and filter higher order function constructs, each FMF expression still represents just a simple value.

There is one caveat when using an FMF expression: the ManyWorlds compiler must be able to derive a finite over-estimation of variable instantiations that are true under the filter. In practice, this often means that the scoped variables occur in some positive Boolean user function application in the filter, with this user function having an enumeration definition that maps to `false` by default. In the above example, the compiler can deduce the finite over-estimation from the user function application `item(x)`.

This overestimation derivation approach is similar to *safe rules* in Answer Set Programming (ASP) [3]. It has the advantage that there is no forced singular finite type associated with each variable. E.g., the following FMF expression asserts that some property (represented by `p`) holds for some triangle in a network (represented by `link`):

```
any[ p(x,y,z) for x,y,z where link(x,y) and link(y,z) and link(z,x) ]
```

When `link` is enumerated as the node-pairs of some large sparse network, it is both cumbersome and inefficient to burden a user to introduce an enumeration of the network's nodes to scope the variables individually.

FMF expressions are originally inspired by conditional quantification in first order logic, where formulas such as $\forall x\colon \varphi(x) \Rightarrow \psi(x)$ represent the truth value that $\psi(x)$ (the map) is true for all (the fold) those x's (the scope) where $\varphi(x)$ (the filter) holds. In predicate logic languages such as ASP [4] and FO($\cdot$) [5], *aggregate* expressions are used for purposes beyond quantification. E.g., the FO($\cdot$) aggregate expression $\mathit{sum}\{\mathit{weight}(x) \mid x \in \mathit{item} : \mathit{inKnapsack}(x)\}$ corresponds to the above knapsack objective expression. Conveniently, FMF expressions provide a streamlined unification of both conditional quantification and aggregates. FMF expressions can be arbitrarily nested, with the common prohibition of variable *shadowing* (rescoping variables already scoped in a parent expression).

3 Accessibility

3.1 Simple yet expressive

A main goal of the ManyWorlds language is to be accessible: the barriers to entry, the obstacles a new user has to overcome before being productive with the language, should be as small as possible. For this, a simple yet expressive syntax and semantics is crucial.

To argue for simplicity, note that Section 2 describes the core concepts of ManyWorlds in only four pages, and most other syntax constructs are merely syntactic sugar on top of this functional core. Simplicity is also the reason why the syntax opts for a low-mathematics style, using keywords such as `implies` and `or` instead of ASCII logical connectives such as `=>` and `\/` – the former are easier to interpret and remember for most people. For the same reason, ManyWorlds sticks with bracket function application notation instead of having a space operator typical for curried functional languages – e.g., `f(x,y)` vs. `f x y`. Most well-known programming languages as well as high-school mathematics use the former, and ManyWorlds wants to spend its *strangeness budget*[4] on other features.

To argue for expressiveness, crucially, in ManyWorlds, one can nest any expression as an argument of another, given that the type of the argument and the nested expression match. E.g., to express the sum of distances in a Hamiltonian cycle (expressed by a `next` user function mapping a city to the next in the cycle) we can write the following:

```
sum[ distance(x,next(x)) for x where city(x) ]
```

This expression nests both `distance` and `next` and can be used anywhere in the program where needed. Note that it does not matter whether `distance` or `next` is known at compile time (via an enumeration definition).[5] The compiler will handle either case appropriately.

As for the expressiveness of FMF expressions, we already argued that it captures quantification and aggregate expressions from predicate logic systems. In addition, the Global Constraint Catalog contains no less than thirteen *joker value* constraints (e.g., `alldifferent_except_0`) [2, 13]. Seen through a functional lens, these are versions of constraints where a fixed filter function ("input is not joker") is applied. ManyWorlds prefers filter functions instead of joker values, as these are more general, and hence, more expressive.

3.2 Debugging

Programming bugs can be roughly divided in three categories:

3.2.1 Compile time bugs

The compiler (or interpreter, grounder. . .) detects that the user has made a mistake. Often, this is a violation of the syntax or typing rules of the language. The ManyWorlds compiler performs such checks taking special care to produce informative error messages. E.g., type checking for scoped variables happens after parsing, at which point the input program string and parse tree are no longer in memory. Still, ManyWorlds will report the error with the line and number of the offending variable(s) in question.

3.2.2 Run time bugs

During execution of a program, an invalid state was reached. Examples are resource acquisition failures (e.g., running out of memory), invalid arithmetic operations (e.g., division by zero), invalid memory accesses etc. ManyWorlds' enforcement of total functions and use of arbitrary precision integer arithmetic ensures any well-formed and well-typed program describes a set of possible worlds and any well-formed and well-typed expression can be evaluated. Hence, any input accepted by the compiler cannot really "fail" and the only run time bugs possible for ManyWorlds are those related to resource acquisition failure. E.g., an out-of-memory error, or the generation of more low level variables or constraints than the solver can handle.

[4] `https://steveklabnik.com/writing/the-language-strangeness-budget`
[5] However, `city` must be enumeration-defined, to derive a finite over-estimation of the filter.

3.2.3 Logic bugs

When a valid program runs fine but produces an unexpected answer, a logic bug is present. In combinatorial programming, these come in two flavors: no solution exists where a user expects one (an *overconstrained* program), and a solution exists where a user expects none (an *underconstrained* program).

3.2.3.1 Overconstrained programs

When the system reports that no world exists, the user can request a set of *blocking constraints*[6] that *together* invalidate all candidate worlds. If the user expects a world to exist, at least one of these blockers does not represent what the user has in mind. Figuring out blockers by hand is pretty cumbersome, so ManyWorlds provides two blocker detection options: *basic* and *detailed*. With basic blockers, ManyWorlds will return the lines and constraints that are causing unsatisfiability. With detailed blockers, ManyWorlds will still return the line numbers of the basic blockers, but will also return simpler constraints that are implied by the basic blockers.

E.g., consider the following unsatisfiable map coloring problem:

```
declare color: string -> {"r", "g", "b"}.
declare border: string, string -> bool.
define border as {("be","fr",true), ("be","lu",true), ("be","nl",true),
    ("be","de",true), ("fr","lu",true), ("fr","de",true), ("lu","de",true),
    ("nl","de",true)} default false.
all[ color(x) != color(y) for x,y where border(x,y) ].
```

The basic blocker option yields `Line 6: all[not color(x)=color(y) for x,y where border(x,y)]`, while the detailed blocker option yields:

```
Line 6: not color("be")=color("de")
Line 6: not color("be")=color("fr")
Line 6: not color("be")=color("lu")
Line 6: not color("de")=color("fr")
Line 6: not color("de")=color("lu")
Line 6: not color("fr")=color("lu")
```

In other words, line 6 contains the culprit, but the problem also lies with countries `"be"`, `"de"`, `"lu"` and `"fr"`, and *not* with `"nl"` (as this country does not occur in the detailed blockers). This way, a user can get a very fine-grained view of the problem.

3.2.3.2 Underconstrained programs

When a world exists that the user did not expect, the user wrote a constraint that is unexpectedly true in the given world. To remedy this, ManyWorlds provides an *evaluation* inference: it constructs a three-valued evaluation of an expression and all its subexpressions in a (partial) world. This provides insight into the inner workings of the constraint, and will hopefully, after some headscratching, point the user to the mistake.

E.g., suppose a user made the common mistake of reversing the material implication:

[6] The blocker set can be minimized when preferred, forming a *minimal unsatisfiable subset*.

```
declare drinksAlcohol: -> bool.
declare age: -> {0 .. 150}.
age() >= 18 implies drinksAlcohol().
```

To their surprise, ManyWorlds may happily oblige with the world

```
define drinksAlcohol() as true.
define age() as 0.
```

Evaluation with the original program extended with these definitions yields:

```
·   ·   age() [0]
·   >= [false]
·   ·   18
implies [true]
·   drinksAlcohol() [true].
```

The evaluation of each (sub)expression is given in square brackets (highlighted in red), the indentation reflects the height in the expression tree. Inspecting this evaluation informs the user why the `implies` statement is not violated, hopefully revealing the mistake.

3.3 Robust compilation

The ManyWorlds compiler follows a *ground-and-solve* (or *flatten-and-solve*) approach. It compiles an input program to an integer program, which is then solved by the low level solving, optimization, propagation and counting routines of the integer programming solver Exact [8].[7] This compilation exploits functions with an enumeration definition to recursively simplify subexpressions and avoids creating auxiliary variables whenever feasible.

For instance, all of the following high-level knapsack constraints are compiled (assuming appropriate declarations and definitions for the knapsack problem were given) to an identical, single 0-1 knapsack linear inequality:

```
sum[ weight(x) for x where item(x) and inKnapsack(x) ] <= 5.
sum[ weight(x) if inKnapsack(x) else 0 for x where item(x) ] <= 5.
sum[ weight(x) * inKnapsack_01(x) for x where item(x) ] <= 5.
```

with `declare inKnapsack_01: string -> {0, 1}` as the 0-1 integer version of `inKnapsack`.

Another example is the compilation of `distinct` (ManyWorlds' alldifferent). Two potential encodings are often employed: an at-most-one encoding and a pairs-of-disequalities encoding. ManyWorlds decides the encoding based on the type of the map expression (`string` is better suited for at-most-one encoding than `int`), the number of subexpressions, and the size of their ranges. The philosophy is that in an accessible system, a (potentially novice) user should not face such decisions – the compiler should bear this burden.

4 Advanced language features

4.1 Syntactic sugar

To streamline common expression patterns, programming languages introduce shorthand notations, and ManyWorlds is no exception. We have already seen the notation `{0 .. 3}`,

[7] Exact was a top performing solver at the 2024 pseudo-Boolean competition (see https://www.cril.univ-artois.fr/PB24/). The compiled integer program requires no specialized propagators so other integer programming solvers could feasibly be used as a backend.

which is syntactic sugar for `{0, 1, 2, 3}`. For string ranges, `{"i" 0 .. 3}` can be used instead of `{"i0"␣ "i1", "i2", "i3"}`.

FMF builtin functions `all` and `any` represent universal and existential quantification, and it is often more readable to write them as quantifications. E.g.,

```
forall x,y where border(x,y): color(x) != color(y)
exists x,y,z where link(x,y) and link(y,z) and link(z,x): p(x,y,z)
```

are syntactic sugar for

```
all[ color(x) != color(y) for x,y where border(x,y) ]
any[ p(x,y,z) for x,y,z where link(x,y) and link(y,z) and link(z,x) ]
```

To denote that a tuple of expressions can take any value from a finite list of value tuples, ManyWorlds provides the `in` notation as syntactic sugar. E.g.,

```
x,y in {(0,"r"), (1,"g"), (2,"b")}
```

is an alternative for the more cumbersome

```
(x=0 and y="r") or (x=1 and y="g") or (x=2 and y="b")
```

Finally, set of known values or value tuples is expressed as a declaration of a Boolean function followed immediately by its enumeration definition with `default false`. ManyWorlds provides `decdef` notations to compact these. E.g.,

```
decdef border as {("be","fr"), ("be","lu"), ("be","nl"), ("be","de"),
    ("fr","lu"), ("fr","de"), ("lu","de"), "nl","de")}.
```

is shorthand for

```
declare border: string, string -> bool.
define border as {("be","fr",true), ("be","lu",true), ("be","nl",true),
    ("be","de",true), ("fr","lu",true), ("fr","de",true), ("lu","de",true),
    ("nl","de",true)} default false.
```

The type signature `string, string -> bool` and `default false` are automatically inferred.

4.2 Intensional definitions

Enumeration definitions are *extensional*: they fix the meaning of a function by listing the exact input-output pairs of the function. This is useful for input data, but less useful to introduce intermediary concepts that may not be fixed by the input data. For this, ManyWorlds allows **intensional definitions** that fix the meaning of a function by describing it using regular ManyWorld expressions. A simple example:

```
declare carried_heavy: string -> bool.
define carried_heavy(x) where item(x) as
    inKnapsack(x) and weight(x) > 3
    default false.
```

This declares and defines the set of carried heavy items as those that are in the knapsack and have a weight greater than three. The user function `carried_heavy` can be used in other expressions without limitation, but its value in a world must satisfy the given definition.

The *head* of an intensional definition (here `carried_heavy(x)`) denotes the user function that is being defined. The head also brings into scope a fresh variable for each input argument of the function. The *body* of an intensional definition occurs after the `as` keyword and denotes the expression to which the head user function is equivalent. The `where` clause restricts the set of inputs for which the definition's body applies. For all other inputs, the definition fixes the defined function's output to the `default` value, ensuring the definition is total.[8] Similar to FMF expressions, the ManyWorlds compiler must be able to derive from the `where` clause a finite over-estimation of the instantiations for the variables that were brought in scope. As a result, only a finite amount of inputs will yield a non-default value for an intensionally defined user function.

Intensional definitions are useful to define auxiliary symbols or derived concepts, that typically have a well-understood meaning in a user's problem domain.

A user function defined by an intensional definition has the crucial property that (when all the user functions in its body are defined) it has **exactly one** possible set of input-output tuples that satisfy the definition. As a consequence, an intensional definition in isolation can never yield unsatisfiability, and any program consisting purely of definitions for all functions allows a single unique world. Informally, a definition *fixes* a user function to *be* its body expression. This property distinguishes definitions from constraints (which can introduce unsatisfiability) and it allows more efficient algorithms under the hood. E.g., when intersecting or enumerating different worlds of a program, it suffices to search for worlds that differ only on undefined functions, as any defined function will not invalidate a found world.

As in other programming languages,[9] ManyWorlds allows at most one definition for each declared user function. ManyWorlds currently also does not allow (indirectly) recursive definitions at ground (flattened) level.[10] E.g., the following is prohibited:

```
declare f, g: -> {0 .. 10}.
define f() as g().
define g() as f().
```

But the following famous definition is allowed as it is not recursive at ground level:

```
declare fib: int -> {0 .. 1e21}.
define fib(x) where x in {0 .. 100} as
    0 if x = 0 else
    1 if x = 1 else
    fib(x-1) + fib(x-2)
    default 0.
```

ManyWorlds' intensional definitions are a generalization of the *predicate* definitions found in FO(·) [5]. Predicates can be viewed as Boolean functions, and for those, a semantics such as the *well-founded semantics* resolves the intricacies with ground-level recursion [7]. For ManyWorlds' function definitions, the semantics is less clear, and ManyWorlds postpones ground recursion support until we have a better understanding of the algorithms involved.

[8] For definitions of constant functions, the `default` clause is optional.
[9] E.g., `https://en.wikipedia.org/wiki/One_Definition_Rule`.
[10] A compiler error for ground level recursion is still under development.

4.3 User types

ManyWorlds has three primitive types – `int`, `bool`, `string` – to declare functions with. `int` and `string` represent infinite sets of values, which means that function application expressions can take any integer number or any string of characters as arguments. Sometimes, this is desirable. E.g., the above Fibonacci definition can have `fib(x-1)` as subexpression without the compiler complaining that `x-1` may evaluate to a negative number.[11]

However, this infinite choice of valid function arguments yields a class of bugs that is hard to detect: typos in strings. E.g., `color("1u")` replaces the letter 'l' in "lu" (representing the country Luxemburg) by the digit 1 (one). The ManyWorlds compiler does not mind: it is a valid input to the function `color`. But it probably is not what the user meant...

To prevent these, ManyWorlds allows to provide *user types* in function declaration signatures instead of the regular primitive types. A user type is a Boolean unary function, with an enumeration definition that defaults to `false`, which represents a finite set of values. Those values are the expected input for the function declaration with the user type, and ManyWorlds will emit a warning should it ever apply the function to a value outside of its user type during compilation. E.g.,

```
decdef country as {"be", "nl", "lu", "fr", "de"}.
decdef rgb as {"r", "g", "b"}.
declare color: country -> rgb.
color("1u")="g".
```

states that `color` expects only `country` strings as input and `rgb` strings as output (the primitive input and output types `string` are automatically inferred). Running this program now helpfully yields:

```
WARNING Encountered function application color("1u") but "1u" does not
belong to input user type country of color
```

In addition, user types provide a form of documentation of function declarations: `color` represents an assignment of rgb values to countries. This implicit domain knowledge can be formally expressed in the program with user types.

5 Inferences

ManyWorlds follows the *Knowledge Base Paradigm* [18] by stressing that each program represents a set of different worlds that a user can interact with. For this interaction, it provides a set of *inferences*: operations that calculate some property of the program and the set of worlds it represents.

The most common inference is *finding* a world (which satisfies all constraints).

Equally common is *optimization*: finding a world that is minimal or maximal under some objective. The objective is an integer expression preceded by the keywords `@minimize` or `@maximize`, as already exemplified by the knapsack example in Section 2.3.

Third, *intersect* yields the intersection of all worlds of a program, which is the set of input-output tuples for declared functions which all worlds for the program share. This inference could equally well have been called "logical consequence" or "full propagation", as it provides all implications of a set of constraints.

[11] Note that `fib(-1)` is defined as `0`, as the unsatisfied `where` clause yields to the `default 0`.

Fourth, *counting* the number of worlds is a classic inference. ManyWorlds extends counting a bit further by also collecting the distribution of some objective amongst the set of possible worlds. The keyword in front of this statistics objective is `@mode`.[12]

We showcase `@mode` by analyzing the odds of a dice problem. Suppose we cast five regular dice with one to six dots on each side. What are the odds of having a total of 14 dots while at most two dice have the same number of dots? And what is the most common value of the largest die in such cases? The following program encodes these questions:

```
decdef die as {"d" 1 .. 5}.
decdef dots as {1 .. 6}.
declare roll: die -> dots.

sum[ roll(x) for x where die(x) ] = 14.
forall y where dots(y): count[ roll(x) = y for x where die(x) ] =< 2.

@mode max[ roll(x) for x where die(x) ].
```

Running the count inference prints:

```
7776 candidate(s) exist
450 world(s) exist
-> 5.787037% of candidates

5: 210 (mode objective fixed to this)
6: 150
4: 90
mean:   5.133333 (77/15)
median: 5
```

So we know that in about 5.8% of all possible die rolls, the constraints are satisfied, and 5 is the most common value for the largest die in those cases. We get even more information: the distribution table for the objective, its mean, and its median.

The last two currently implemented inferences are those used for the debug functionality: *explaining* unsatisfiabile programs and *evaluating* expressions in a partial word can be used for tasks other than debugging as well.

6 Related work

ManyWorlds' main inspiration is FO(·) [5]: an extension of typed first order logic with arithmetic, aggregates and inductive definitions. FO(·) and ManyWorlds share an adherence to the *Knowledge Base Paradigm* [18] where a specification is seen purely as a description of a set of possible states (worlds), and a range of different computations (inferences) can be carried out over a single specification. ManyWorlds aims to be a simpler, less mathematical and more approachable language than FO(·), using functions as basic building block instead of FO(·)'s relations. This focus on functions instead of relations also is the main difference between ManyWorlds and the high-level, solver-agnostic combinatorial programming language ESRA [9].

[12] https://en.wikipedia.org/wiki/Mode_(statistics)

Concerning functions, the Satisfiability Modulo Theories (SMT) standard SMT-LIB [1] features *uninterpreted functions* as core building blocks. SMT-LIB is an expressive language, allowing even potentially infinite co-domains and variable ranges. It is, however, not accessible to a programmer without a formal mathematical background, if only because writing basic arithmetic in the enforced prefix notation takes quite some getting used to.

Essence [10], EssencePrime [14] and MiniZinc [12] are high-level constraint programming languages with which ManyWorlds shares similarity. From a pure expressiveness viewpoint, ManyWorlds brings little new to the table. E.g., Essence supports enumerated types (covering the use case of string values in ManyWorlds), EssencePrime supports matrix comprehension, and MiniZinc supports enumerated types, set and list comprehensions, and a ternary conditional.

Instead, ManyWorlds distinguishes itself by a focus on accessibility. It aims to be more abstract and to avoid the need to learn about global constraints, variable arrays, matrices, indices, lists, propagators, search heuristics etc. It minimizes the difference between parameters and decision variables, with the main remnant being the requirement to provide a derivable finite over-estimation of instantiations for variables scoped in a `where` clause. ManyWorlds replaces most mathematical notation by natural language connectives, lowering the barrier of entry for non-technical users. ManyWorlds also comes "batteries included" with strong debugging support using fine-grained blocker computation and recursive expression evaluation, and with multiple generic inferences at the touch of a button. In short, ManyWorlds aims to be a very humble Python to MiniZinc's and Essence(Prime)'s powerful C++.

7 ManyWorlds for a MiniZinc use case: on-call rostering

A detailed language comparison to MiniZinc or Essence would go beyond the scope of this paper. Instead, we program the on-call rostering problem from the 2018 MiniZinc challenge[13] in ManyWorlds and remark on the most striking differences.

The on-call rostering problem consists of assigning weekdays to a single surgeon who is on call for that day – this assignment is called the *roster*. The surgeon assigned to a Friday is also on call for the whole weekend, so Fridays are treated special and Mondays are considered to immediately follow Fridays. Surgeons cannot be on call on consecutive Fridays or three days in a row, and those on call on Thursdays or Mondays should not be on call on the Friday in between. Finally, surgeons can input days when they are unavailable and can manually fix days in the roster (which overrides all previous hard constraints). The preferred roster minimizes the number of adjacent days with the same surgeon, the number of Wednesdays that have the same surgeon as the following Friday, and the difference in on-call work load between surgeons.

We picked the on-call rostering problem because of its functional core – the roster is simply a function mapping days to surgeons – and two input properties are naturally expressed as functions as well – days are mapped to their weekday name and surgeons to their work load commitment. A cleaned up and commented ManyWorlds on-call rostering program is available online.[14] We copied the variable names of the original MiniZinc program as user function names in the ManyWorlds program to make comparison easier.

[13] `https://github.com/MiniZinc/mzn-challenge/blob/develop/2018/on-call-rostering/oc-roster.mzn`
[14] `https://tinyurl.com/manyworlds-on-call-roster`

Observation 1. All concepts in the on-call rostering problem can be represented by functions:

- properties such as the total number of weekends and week days each surgeon is on call – `week_days_oc`, `weekend_days_oc`
- sets such as the pool of surgeons or the list of days – `staff`, `days`
- relations such as the fixed and unavailable days – `fixed`, `unavailable`
- constants such as the number of staff and number of days, the minimization weights for soft constraints, and extra terms in the objective function – `num_staff`, `num_days`, `adj_days_str`, `wed_before_weekend_str`, `week_day_bt`, `week_day_bt`

Observation 2. The MiniZinc specification does not explicitly introduce the weekday names, even though these feature prominently in the problem domain: Mondays, Tuesdays, Wednesdays, Thursdays and Fridays are all handled differently. On this front, ManyWorlds' string type allows a close translation of the problem domain description, increasing readability of the constraints and confidence of the programmer in the correctness of their code.

Observation 3. The MiniZinc specification contains sanity checks using the special `assert` notation (lines 114 to 155). These are undoubtedly useful, and the ManyWorlds specification mirrors these as regular constraints. Should these fail, the program will be unsatisfiable and a user can just calculate the basic blocking constraints. These will point to the line of the offending constraint, telling the user what the problem is. An informal explanation of each input-checking constraint is added as a comment the line, taking over the role of the information string in the MiniZinc `assert`.

Observation 4. The MiniZinc specification introduces auxiliary variables `week_days_oc` and `week_days_oc` to more easily write the complex work load balancing constraints. Auxiliary variables are a common modeling practice and often they match sensible concepts in the problem domain with well-understood meanings. ManyWorlds' intensional definitions are an elegant way to specify auxiliary concepts: the defined function represents the concept, and the body of the definition represents the meaning of the auxiliary concept in the problem domain. In the ManyWorlds on-call rostering program, the intensional definitions for `week_days_oc` and `week_days_oc` exemplify this.

Observation 5. The minimization objective of the MiniZinc program makes use of further auxiliary variables `adj_days` and `wed_before_weekend` to represent the soft constraints on adjacent days and Wednesdays before weekends. The ManyWorlds program encodes these directly in the objective, making for a pretty complex expression. This is a subjective modeling choice, but it showcases that ManyWorlds can handle deeply nested expressions without hassle.

Observation 6. Of course, other ways of specifying the on-call rostering problem are possible in both ManyWorlds and MiniZinc and **more elegant or more efficient ways probably exist**. In the end, a programmer always has to balance computational efficiency with closely modeling the problem domain. ManyWorlds definitely is geared toward the latter, while the MiniZinc specification allows the opposite: lines 277 to 293 pass hints to the search heuristic of the solver, hopefully improving performance. ManyWorlds currently has no such feature.

7.1 Performance comparison

Though a full performance comparison of ManyWorlds with other combinatorial systems is out of the scope of this paper, we can solve the MiniZinc on-call rostering specification with Gecode [16] and OR-Tools [15] and compare it with ManyWorlds' on-call rostering runtime.

	MiniZinc (Gecode)		MiniZinc (OR-Tools)		ManyWorlds (Exact)	
Instance	Objective	Time	Objective	Time	Objective	Time
2s-200d	65	25 000+	**64**	**6204.30**	**64**	**0.04**
4s-100d	58	25 000+	61	25 000+	**2**	**0.24**
10s-100d-C	47	25 000+	**47**	**1.29**	**47**	**0.06**
20s-100d-B	59	25 000+	58	25 000+	**16**	**0.43**
30s-400d-A	293	25 000+	299	25 000+	**2**	**11.12**

Table 1 Objective values and runtimes (in seconds) for three different approaches to solve the on-call rostering problem. Entries in bold denote that optimality was proven.

We run Gecode version 6.2.0, OR-Tools version 9.1.9490 and ManyWorlds commit 78bd63d3 on an AMD 5950X machine with 32 GiB of RAM and a timeout of 25 000 seconds. The five instances on the MiniZinc Challenge repository are used.[15] Run scripts and specifications are available online.[16] All time data includes compilation (flattening) time, which was insignificant relative to the solve time. Table 1 presents the results.

For these five optimization problem instances, ManyWorlds performs great!

8 Conclusion and Future Work

ManyWorlds is a new combinatorial programming language that aims to be accessible by using functions as its fundamental building block.

The number of supported features is steadily rising, but a lot are still missing. Crucially, the performance characteristics of ManyWorlds are not yet established either. In the long run, the goal is to have "good enough" performance on finding (optimal) worlds and to have "first-in-class" intersect and blocker calculation performance.

The latter are crucial in interactive configuration settings [18] where a user iteratively fixes a partial world and the system provides feedback in the form of logical consequences and explanations for those derived consequences. For this use case, we also envision extending ManyWorlds with a *relevance* inference [11], which computes the ground function symbols that can still contribute to the satisfaction of a constraint. Here, the property that each definition fixes a function to one unique interpretation is again important.

On the syntax front, full support for *tuple expressions* would be great to have, as well as a form of definition that calls Python routines lazily instead of the eager Python enumeration definitions. The latter would allow a user to elegantly call string, date or mathematical library functions from Python without having to precompute all possible inputs. Next, a stateful interface for ManyWorlds would allow repeated calls and the combined use of different inferences – the backend solver Exact already provides the necessary stateful solving routines.

Finally, the practical usability of ManyWorlds should be further established. It is worth noting that already one major project with ManyWorlds was completed: a worst-case analysis of three different electoral systems applied to the recent Belgian elections.[17] More is – hopefully – to come!

[15] `https://github.com/MiniZinc/mzn-challenge/tree/develop/2018/on-call-rostering`

[16] `https://gitlab.com/nonfiction-software/manyworlds/-/tree/main/examples/on_call_rostering_scripts?11d91404`

[17] `https://jodevriendt.com/belgian-elections`

References

1 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

2 Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12:21–62, 2007. URL: `https://api.semanticscholar.org/CorpusID:14681654`.

3 Pedro Cabalar, David Pearce, and Agustín Valverde. A revised concept of safety for general answer set programs. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 58–70, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

4 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *TPLP*, 20(2):294–309, 2020. `doi:10.1017/S1471068419000450`.

5 Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modeling language: the IDP system. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 279–323. ACM / Morgan & Claypool, 2018. `doi:10.1145/3191315.3191321`.

6 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

7 Marc Denecker and Joost Vennekens. The well-founded semantics is the principle of inductive definition, revisited. *KR*, pages 22–31, 01 2014.

8 Jo Devriendt. Exact solver, 2023. URL: `https://gitlab.com/nonfiction-software/exact`.

9 Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation*, pages 214–232, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

10 Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2007. URL: `https://api.semanticscholar.org/CorpusID:1931705`.

11 Joachim Jansen, Bart Bogaerts, Jo Devriendt, Gerda Janssens, and Marc Denecker. Relevance for sat(id). In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 596–602, United States, 2016. AAAI Press. International Joint Conference on Artificial Intelligence, IJCAI ; Conference date: 09-07-2016 Through 15-07-2016. URL: `http://ijcai-16.org/index.php/welcome/view/home`.

12 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

13 Nicolas Beldiceanu and Mats Carlsson and Sophie Demassey and Thierry Petit. Global constraint catalog joker value constraints. `https://sofdem.github.io/gccat/gccat/Kjoker_value.html#uid7497`, 2014.

14 Peter Nightingale and Andrea Rendl. Essence' description. *ArXiv*, abs/1601.02865, 2016. URL: `https://api.semanticscholar.org/CorpusID:17820419`.

15 Laurent Perron and Frédéric Didier. Cp-sat. URL: `https://developers.google.com/optimization/cp/cp_solver/`.

16 Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with gecode, 2010. URL: `http://www.gecode.org/doc-latest/MPG.pdf`.

17 The Coq Development Team. The Coq reference manual – release 8.19.0. `https://coq.inria.fr/doc/V8.19.0/refman`, 2024.

18 Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, and Marc Denecker. The KB paradigm and its application to interactive configuration. *Theory Pract. Log. Program.*, 17(1):91–117, 2017. `doi:10.1017/S1471068416000156`.