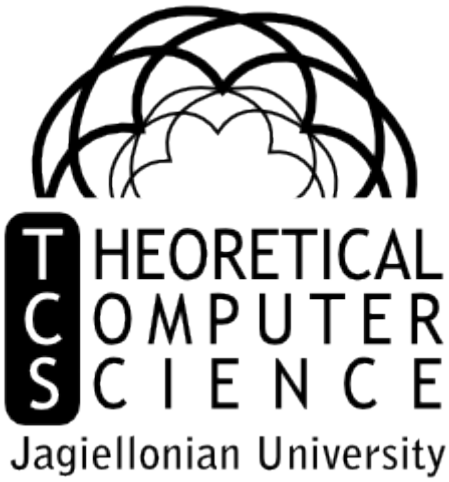

Master Thesis

Maciej Poleski

Designing a hybrid declarative-imperative programming language

Supervisor: dr. Grzegorz Herman

Kraków 2017

Table of Contents

1. Preface

Software engineering using imperative languages often leads to the necessity of implementing solutions of simple problems. Defining such a problem in a declarative manner and leaving its solution to the compiler may reduce the amount of work necessary to create software, and potentially avoid programming errors created during imperative implementation.
Possibility of leaving more work to the compiler may be valuable also in cases when the compiler is unable to solve the problem efficiently - if the gain from reducing development work necessary to solve the problem is greater than the "losses" from lower performance of the solution. Such situation might take place during prototyping, or with rarely used functionality.

Prolog is a well known, Turing-complete computation model, and it can be implemented effectively.
Loops and conditional expressions are the building blocks of explicit control flow in imperative languages. These constructs make even simple programing languages Turing-complete.

The goal of this work is to explore the possibility of merging declarative and imperative semantics in one programming language.
I investigate the feasibility of merging the key properties of Prolog with constructs from imperative languages (such as loops, conditional instructions and functions), and a create programming language featuring this merge.

Additionally, I would like to enable linking modules written in this new hybrid language with modules created using existing imperative languages. As a compatibility layer, the standard x86_64 ABI for libraries written in C on the Linux operating system[1] should be used.
Another goal is to explore potential usages of such language. The possibility to link with modules written in other languages (C and C++) allows to incorporate functionality based on declarative implementations to existing projects.

A complete compiler has been implemented and its source code made available at https://github.com/Maciej-Poleski/kompilator.

1 https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-r252.pdf

2. Language

2.1. Syntax

The syntax of the language is influenced by C and Prolog. Source code consists of record definitions and predicate definitions (equivalent of C functions). Most of constructs used in imperative predicate definitions look exactly as in C:

```
void f() {
  int i;
  var v;      // fresh unbound variable
  *int ptr;   // unbound pointer variable, similar to int* ptr in C
  ptr = &i;   // assigns address of i to ptr
  for(i = 0; i < 10; i=i+1) {
    if(i==5) {
      break;
    } else {
      continue;
    }
  }
  v === ptr; //  unifies v with ptr, variable v is bound to term ptr
}
```

A unification expression makes two terms equivalent. This is different from assignment (copying - detaching), because unified terms are the same forever (modification of one is immediately reflected in the other).

Predicates can also be defined declaratively, using the operator ":-". Each term can be referenced later using its name. There are no constants, but function symbols can have arity 0. Note the semicolon (`;`) at the end of definition:

```
add(z(), var Y, Y) :-;
add(s(var X), var Y, s(var Z) :- add(X, Y, Z);
```

There is also an alternative syntax for predicates with explicit output argument. Output argument can be referenced using the `$` operator:

```
var Y = add(z(), var Y) :-;
s(var Z) = add(s(var X), var Y) :- eq(Z, $add(X, Y));

eq(var X, X) :-;
```

An imperative definition of similar (but much more efficient) `add` function could look as follows:

```
int Z = add(+int X, +int Y) {
  Z === X + Y;
}
```

Any function symbol can be either a term (equivalent of a value from C) or a predicate call (equivalent of a function call from C). By default, function symbols in imperative predicates are interpreted as calls and in declarative predicates as terms:

```
void f() {
  g();   // function call
}
f() :- g();   // term (goal)
```

Operators `` ` `` and `$` can be used to override these defaults:

```
void f() {
  `g();   // term
}
f() :- $g();   // function call (g() will be called, result value is a goal)
```

Record definitions resemble structures from C language (without trailing semicolon):

```
struct MyStruct
{
  int x;
  int y;
}
```

2.2. Type system

The aim of the type system is to have the same term representation in both imperative and declarative predicates. That enables seamless integration of both kinds of code.

There are three primitive types:

- `bool` (boolean: `true/false`),
- `int` (32-bit signed integer), and
- `var` (variable),

three type constructors:

- `*` (pointer),
- `+` (annotation), and
- `struct` (record),

and one auxiliary type of all function symbols.

An annotated type represents the same set of terms as one without annotation, but does not allow unbound variables nor uninitialized values in any subterm during unification.

Examples

Literals are values of annotated types:

```
true has type +bool
1337 has type +int
```

Assignment (in contrast to unification) requires exact type match:

```
// int x = 2;   // ill-formed - 2 has type +int
+int x = 2;
```

Address-of operator preserves annotations:

```
// *int ptr = &x;   // ill-formed - &x has type *+int
*+int ptr = &x;
```

Annotation enforces initialization of subterms during unification:

```
struct S {
  int x;
  +bool y;
}
S s;
s.y = false;
// +S t = s;   // ill-formed - type mismatch
+S t === s;   // well-formed, will fail - s.x is uninitialized but t is annotated
s.x === 5;
+S u === s;   // u.x == 5
```

2.3. Unification

A unification expression binds two terms, making them equivalent (or the whole operation fails):

```
var X === f(1, var A);
var Y === f(var B, 2);

X === Y;
// Terms X and Y share their information
```

```
// A == 2; B == 1;
// X == Y == f(1, 2)
```

Unified terms share their structure - modification of one is reflected in the other.

```
var X;
var Y === f(X);
var Z === f(Y);

// Z == f(f(X));

X === g(2);

// Y == f(g(2));
// Z == f(f(g(2));
```

The unification fails if common information contradicts:

```
1 === 2;  // fails, 1 != 2
```

Unification cannot be used to create infinite terms:

```
var X === f(X); // fails, term f(X) cannot be unified with X because it has X as a
                // subterm
```

2.4. Program execution

Each source file of the program (translation unit) is a set of record definitions and an ordered list of predicates. Only predicates are executable (become part of a module).

The program state consists of a substitution (a mapping of bound variables to terms), goals (a list of terms), and rollback stack (record of changes to both the substitution and goals, designed to enable rolling changes back). Predicates can modify program state by adding new mappings to substitution and new goals to the list.

Predicate execution

Each attempt to execute a predicate `t` follows the procedure described below:

```
// store current context to enable rollback in case of failure
old_context = current_context;
for(predicate p: module) {
  // create a new context based on the current one (initially
  // the same)
  new_context = current_context;
  // try to unify t with p, modifying new_context
```

```
  if(unify(t, p, new_context) == SUCCESS) {
    current_context = new_context;
    // execute p in new context
    status = execute p;
    if(status == SUCCESS) {
      return SUCCESS;
    }
    // restore the context from before unification
    current_context = old_context;
}
return FAILURE;
```

If any subexpression within a predicate fails, then the whole predicate does. When predicate finishes execution successfully, it additionally makes an attempt to execute the first goal from the list (or succeeds immediately if there are none). The goal is removed from the list in the context of finishing predicate.

2.5. Example

The following code can be used to check if, in a given graph (DAG), there exists a path between two given vertices. Vertices are identified using integers.

```
// number of edges outgoing from vertex
int r = edges(+int vertex) extern "C"

// edge outgoing from vertex, 0 <= index < edges(vertex)
int r = edge(+int vertex, +int index) extern "C"

// check if a path from v to u exists
path(int v, v) :- ;
path(int v, int u) :- edge(v, u);

path(int v, int u) :- iterate(v, u, 0);

iterate(int x, int y, int i) :- lt(i, $edges(x)),
                                path($edge(x, i), y);
iterate(int x, int y, int i) :- lt(i, $edges(x)),
                                iterate(x, y, $inc(i));

lt(+int a, +int b) {
  if(a>=b) {
    fail();
  }
}

int r = inc(+int a) {
  r === a+1;
}
```

The first two predicates are declared as “C” language functions. Their implementation will be necessary during program consolidation (linking). Example implementation in C:

```
int edges(int vertex) {
  return vertex;
}

int edge(int vertex, int index) {
  return index;
}
```

The `path` predicate is a declarative implementation of (naive) graph traversal algorithm. First two variants are the corner cases. The third case delegates to `iterate` helper in order to iterate through all edges outgoing from `v`.

The first `iterate` variant tries to prove that `i` is less than the number of edge outgoing from `x`, and that there exists a path from `x` to `y` through the `i`-th vertex on `x`’s neighborhood list. The expression `$edges(x)` is a call of predicate `edges` with argument `x`. Result of the call, say `n`, becomes part of the enclosing term `lt(i, n)` and becomes the first subgoal to be proven (this is the loop condition). The second subgoal is created by calling `edge(x, i)` to get the `i`-th neighbor of `x`. If both goals are proven (we did not exceed the length of the neighborhood list, and did find a path from a given neighbor to `y`) we are done (predicate succeeds).

Otherwise, the second `iterate` variant redoes the loop condition check and creates a new `iterate` goal with `i` increased by one. This “recursion” facilitates looping over `x`’s neighborhood list.

The last two predicates are utilities:

- `lt` checks if a less than condition is *not* met and fails in such case. Failure is triggered by an attempt to call a predicate which does not exist. Note the presence of type annotations in arguments. It is necessary for both integers to be in defined (set) state for the comparison operator. If an attempt is made to call this predicate with an integer in undefined (uninitialized) state, the call will fail (and the next predicate in order will be attempted). When predicate execution reaches the end of code, execution is considered successful.
- `inc` performs the incrementation. A named argument placed on the left of ‘=’ can be used exactly as other arguments. Note how this “result” is returned by unification. This is also the implicit term embedded in place of a call.

3. Compiler

I implemented a classic ahead-of-time compiler, translating from plain text source code to object code. It currently targets only x86_64 GNU Linux, but can easily be modified to emit code for other platforms supported by LLVM. Due to the dynamic nature of the language, some tasks performed by the compiler in other languages (for example C) are moved to runtime.

3.1. Parsing

The task of parsing is performed using two classic supplementary tools: Flex and Bison.

Flex is used to generate the scanner. This process takes place during compiler compilation. The character set used by the generated scanner is the one used during scanner generation - note that this does not give a fully predictable outcome (depending on user environment it can be UTF-8, but can also be a different encoding). This is why the language does not have a built-in character type.

Bison is used to generate the parser. Even though Bison is advertised as a GLR parser generator, GLR + C++ + variant is not supported[2]. This reduced the ability to support complex structures, and required some workarounds in the grammar specification to make it LALR-compatible.

3.2. Name resolution

It is not necessary to use forward declarations of entities defined within the same translation unit. Function calls are resolved at runtime - these are out of scope for name resolution.

Name resolution performs two passes over each translation unit. The first pass collects definitions of structures. The second pass resolves both type names (using data from the first pass) and variable names. Variable names are always defined before use. There are no global variables.

3.3. Code generation

The compiler uses LLVM as a backend to emit object code. It is currently fixed to x86_64 GNU Linux target, but can be easily modified to add other targets supported by LLVM.

All declarative definitions are translated to code and emitted in form indistinguishable from imperative definitions. This means that no overhead is incurred due to actual choice of function definition style.

The generated code uses a special runtime library, shipped with the compiler. This library is responsible for maintaining program state, including term structure, goal list and rollbacks.

2https://lists.gnu.org/archive/html/bug-bison/2015-03/msg00003.html

One interesting property of the compiler's design is that the representation of terms is not known to the compiler. All operations on terms are performed through runtime. This can be seen as an abstract interface, enabling use of different runtime implementations, depending on needs of specific applications.

4. Runtime support

The runtime is a library provided with the compiler. Each application using modules compiled by this compiler has to be linked with this library.
It consists of a set of functions and objects designed to maintain and manipulate program state including terms and goals. Each modification of program state can be rolled back if necessary. The runtime is also responsible for all dynamic operations on terms, including unification, calls, and term comparison.

4.1. Term representation

Representation of terms is based on "implicit term representation" from http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf, augmented with fields necessary to store the value depending on the type of the term.
All terms are allocated on the heap, with their lifetime managed in deterministic manner.
Term representation consists of type information and value. Type information is always at the same constant offset from the beginning of the term - this is necessary to access the type information of unknown terms.
Value depends on the type and can be:

- for `var` it is a pointer (UnknownBlob*),
- for `int` it is `int32_t`,
- for `bool` it is `bool`,
- for pointer it is a pointer to pointee,
- for function symbol it consists of the arity, name, and pointers to subterms,
- for record it consists of the name of the record type, and pointers to field values.

4.2. Unification

The unification algorithm used by the runtime is an extended version of the standard first-order unification algorithm. Recursion is forbidden. The extension is necessary to handle data types with values and annotations. These are the rules used by the algorithm:

- An unbound variable can be unified with any term (provided that this variable is not a subterm of this term).
- A function symbol can be unified with another function symbol if both have the same name and arity. Unification recurs over subterms.
- An `int` term without value can be unified with any int term.
- Two `int` terms with values can be unified if their values are equal.
- `bool` terms follow the same rules as `int` terms.
- A pointer with no value (no address set) can be unified with any pointer of the same type.
- Two pointers with value (set address) can be unified if both point to the same object or both are null (and of the same type).
- A record can be unified with another record of the same type. Unification recurs over fields.
- All the other combinations of terms cannot be unified.

If the algorithm succeeds, an additional property is checked. All terms of annotated type (and all their subterms) shall have a value. Unbound variables are considered as having no value. If this requirement is not met, unification fails.

4.3. Application state

Application state consists of three data structures:

- Term graph - all the terms and relationships between them. Representation of single term is described in more detail in “Term representation” chapter. This graph is acyclic.
- Goal list - a list of terms. It is a queue of predicates which should be executed in order.
- Rollback stack - a stack of callables, used for backtracking.

Program execution starts when the first predicate is called. It can be either the `main()` function or the call may originate from other parts of the application.
Predicate execution can fail. Failures can originate from two sources:

- Unification of some expressions failed
- Attempt to call another predicate failed

When it happens, all mutations performed by that call are rolled back.
Predicates can modify program state in the following ways:

- Add a new goal to the back of goal list.
- Unify two expressions. If unification fails, so does predicate execution.
- Attempt to call some predicate.
  When an attempt is made to call a predicate `p`, all defined predicates with heads unifiable with `p` are considered eligible for a call. For example for the call `f(1, false)` these predicates are considered eligible:
  ```
  f(var X, var Y) :- ...
  void f(var X, +bool b) { ... }
  int i = f(+int a, bool b) :- ...
  ```
  and these are not eligible:
  ```
  g(var X, var Y) :- ...
  f(int a, int b) :- ...
  ```

  Eligible predicates calls are attempted in order of their definition in the module. If any call succeeds, control flow returns immediately to the call site. If all eligible predicate calls fail, so does the call.
- Assign a value to a pointer. Pointer is the only mutable data type. Assignment to the pointer is reflected in all the other pointers unified with it.

4.4. Modules and C interoperability

Each source file is compiled into a single module. A module consists of ordered list of predicates (data type definitions are not part of the module).
Predicate can be:

- declarative
- imperative
- extern
- extern “C”

Declarative predicates

Declarative predicates have the form

```
f(var X) :- g(X, 1337);
```

When executed, a list of specified goals (in blue) is appended to the goal list and evaluation continues by taking the first goal from the goal list.

Imperative predicates

Imperative predicates have the form

```
int i = f(+int a) { i === a+1; }
```

Execution doesn't directly add any new goal to the goal list, but can call declarative predicates to do so. After finishing execution, evaluation continues by taking the first goal from the goal list.

Extern predicates

```
f(+int a) extern
```

Their real definition (either declarative or imperative) is in another module. All defined predicates are exported, which allows calling them from other modules. The only requirement to use a predicate from another module is to declare it as extern.
The program is ill-formed if two modules export the same predicate (with the same head).

Extern "C" predicates

```
int i = f(+int a) extern "C"
```

Their real definition is in another module and uses "C" language calling convention. This functionality allows using modules implemented in other languages.
For each extern "C" predicate, the compiler generates a stub function which is called exactly as normal predicates. The stub translates terms representation to native C ABI representation for the function call. The return value is translated back from C to term representation, and unified with the designated result term (if the function is non-void). Such predicate calls always succeed.

Exporting predicates to "C"

All non-extern "C" predicates are also exported using C ABI, provided that they have no variables or function symbols as arguments, and that all the arguments (not including the result argument) are annotated.
For each batch of predicates (sublist of predicates defined within module with the same head), one stub function is generated by compiler. The stub function uses "C" language calling convention. It translates incoming arguments into terms and attempts to call a predicate. It is assumed that at least one of predicates within a batch will succeed, otherwise the behavior of the application is undefined. Result argument (if present) is translated back to native C ABI representation and returned to the caller.

It is currently not supported to import/export from/to "C" functions with pointers as arguments. This functionality can be added with big cost independently or for free using annotated term representation optimization described in more detail in the "Possible extensions" section.

5. Possible extensions

5.1. Optimization of predicate call by selecting predicate implementation during compilation

In general, it is impossible (the problem is undecidable). In many cases (especially for code written in imperative style) it might be easy to prove that "if this call succeeded, then it must have been to "that" predicate implementation". In other cases, it could be possible to reduce the set of possible predicate implementations whose call might succeed.

5.2. Defining new declarative predicates at runtime (like asserta/assertz)

It can be implemented in at least two ways:

- By implementing an interpreter and registering a new predicate (+ information about the interpreter) in the module,
- Using Just-in-time compilation (which is supported by LLVM) and registering in the module.

Note that the ability to define new predicate on runtime disables the ability to choose "the one" implementation during compilation (because another compatible predicate can always be added just before call). It might be worth analyzing restricted variant of runtime predicate definition. For example, if we limit the ability to define predicates on runtime to predicates with head of constant (known during compilation) function symbol (for example `my_function(X, Y, Z) :- …`, where `my_function` is given literally in source code, and `X`, `Y` and `Z` are, possibly bound, variables), then we will be able to "blacklist" specific calls from early-binding, and proceed with early-binding of all other calls. In this example, the calls to `my_function(A, B, C)` would never be early-bound, while the calls to `my_function(A, B)` could still be.

5.3. Unification during compilation

Terms without variables can be unified by the compiler (because their structure is known during compilation). This feature can further improve "early-binding predicate call". Note that such unification will always be possible for pure imperative functions (as their argument structure is fully known during compilation).

5.4. Optimized representation for annotated types instances

Annotated terms cannot exist without full expansion and all values. In such case, preserving term identity during unification is not required for algorithm correctness. That enables the possibility to move annotated term representation to the system stack.

When the full extension of a term is known during compilation (which is the case for example in pure imperative functions), its representation can be flattened and moved to the stack without dynamic type information. That makes representation of imperative types the same as representation of C counterpart type. That, combined with two optimizations described above, would enable emitting code of imperative functions without the overhead of mixed declarative and imperative semantics.

5.5. Other possibilities

From other possibilities, three are particularly interesting: subtyping, array type, and customizing predicate rollback implementation.

There is currently no subtyping in the language. It might be considered how to add subtyping and how unification would look like in such type system.

Array type is also absent. It is different from other types because instance information can be rectified in more steps. From least information to most complete it could be:

- unknown (variable),
- an array (but size and element type unknown)
- array of known size and unknown element type
- array of known element type but unknown size
- array of known size and element type
- and recurrence on elements of array

Note that information about size could possibly be constrained by “size(array) >= 3” and array[2] == 44 – without information if array[3] exists. This is different from existing types which can be either unknown (variable), uninitialized, or initialized (+ possible recurrence on subterms).

Current rollback mechanism is limited for predicates defined within language – it is unable to cleanup side effects caused by calling predicates defined in e.g. C++. One interesting extension to consider is to allow specification of custom cleanup mechanism for predicates. This mechanism could be then used by programmers to fill-in this gap.