The T1 Programming Language

Thomas Pornin

May 14, 2019

Abstract

This document describes the design of the T1 programming language. It is intended to serve both as a specification of the core language features, and a rationale for the design decisions.

T1 is an imperative language that offers memory safety, generic metaprogramming, strong static typing through whole-program analysis, and support for object-oriented programming. It is geared toward efficient support of memory-constrained architecture, and in particular can be embedded as efficient co-routines with tight bounds on stack usage. Nevertheless, T1 aims at also being usable as a general purpose language.

WARNING: T1 is a work in progress, and the specification may be altered as implementation of the compiler unveils unforeseen difficulties or potential desirable feature changes.

1 Overview Of T1

1.1 Project Goals

T1 aims at providing a number of features that are not all obtained together from existing languages. It can be viewed as an extension of T0, the Forth-like language which is used for some parts of BearSSL (namely, for processing handshake messages, and decoding X.509 certificates). Within the context of BearSSL, T0 exists so as to express complex nested decoding and encoding over streamed data: in order to explore an encoded object with nested structures without requiring buffering of the entire object, the decoding process must be interruptible and restartable, i.e. live as a *coroutine* that can be scheduled to run when and only when data bytes become available. The standard C language lacks coroutines; coroutines can be defined in non-standard ways, but will require an in-memory stack of non-negligible size for constrained systems; T0 provides a lightweight coroutine that uses a very small custom stack (with guaranteed limits on maximum stack growth).

T1 is an evolution of T0 into a more ambitious project:

- Like T0, T1 must support compilation into components that can be embedded within applications that run on "bare metal" systems (no OS support).
- Lightweight coroutines must be supported.
- T1 must be (by default) memory-safe. T0 has only limited memory-safety, in that the compiler proves strict bounds on maximum stack usage; however, array accesses in T0 are not checked with regards to array boundaries.
- T1 shall support object-oriented programming. OOP is primarily a way to structure application code; in plain C, this is usually done with function pointers arranged in vtables. The language can provide primitives that help with OOP, for a simpler and safer syntax. T0 has no specific OOP features.
- T1 should *optionally* support dynamic memory allocation. This more or less implies the use of a garbage collector, to maintain memory safety. It must be possible to write code that does not use the GC, and, when the GC is present, it must be configurable with strict limits on allocated size.
- Code written in T1 is meant to be embeddable in applications that primarily use C; thus, call to T1 code from C, and to C from T1, shall be simple and efficient. In particular, the memory layout of T1 objects should have a predictable counterpart in the C world, so that direct access is feasible and easy.
- While T0 has only a single flat namespace, T1 should offer some ways to segment the code space into units that limit the risk of name collision, especially if several different developers are involved.

Memory safety and OOP both require the definition of a rich type system, which may then also be used by the developer to express constraints on the structure of the application, that will be verified and enforced at compilation and/or runtime.

Since T1 aims at being a generic purpose language, it should be possible to write complex applications entirely in T1, starting with the T1 interpreter/compiler itself (as is traditional for language development).

Programming language design includes the syntax, which sits between a rational, deterministic machine (the computer) and a definitely less rational and deterministic human (the developer). As such, the language necessarily has an aesthetic facet, which cannot be rationally argued for or against. As a working principle on these matters, I define myself as the sole judge; T1 should aesthetically please me.

1.2 Main Features

T1 combines many features inspired from other programming languages, but not hitherto found together in a single language. Inspiration has been drawn from, in no particular order, C, Java, C#, Forth, Caml, Rust, Go, and others. Some of the features will be explained in terms of comparisons with these other languages. In this section, we give an overview of the main features.

Imperative. T1 is an imperative language. More generally, it strives to give to the programmer a clear mental picture of what happens in the generated code; notably, order of execution of all operations is duly specified, and should match whenever possible left-to-right reading order in the source code. Automatic optimizations should be kept at a minimum (e.g. no automatic vectorization with SIMD instructions). This can be thought of as a "no magic policy".

Fully Specified. There is no "undefined behaviour". All operations occur in fully specified ways (this is similar to Java, and unlike C). There may be some platform-dependent characteristics, such as the possible range of integer values that can serve as array indices (this corresponds to types such as `size_t` in C, or `usize` in Rust).

Combined Interpretation / Compilation Model. The source code, when processed by the T1 engine, is really executed, as a script. The *compiler* is a final optional phase that can extract some of the defined functions and serialize them into an executable form. While interpretation and post-compilation execution work on the same functions, they use quite different models:

- During interpretation, everything is dynamic. Functions and types can be referenced before they are defined; a violation is reported only when trying to actually call a function which is not yet defined. Interpreted code has full access to API that allow creating new types and functions.

- Compilation performs a thorough static type analysis that will refuse to produce the executable output for code that does not comply to strict rules. The point of the rules is to ensure that execution won't fail with a type-related error; they can also provide guarantees on good memory-wise behaviour. In particular, compiled code is not allowed to be recursive, so that maximum stack usage can be *a priori* bounded.

Processing source code as a script to be executed means that compilation necessarily involves *executing* the source code. The interpreter will provide a specific "sandboxing" mode which will prevent access to system-level features such as the network, or files outside of the source code collection itself. The ability to sandbox potentially hostile code is not considered a primary feature, but this should ultimately be supported.

Extensible Postfix Syntax. T1 uses a Forth-like syntax, which relies on postfix notation: operations appear after the operands. While this syntax does not follow traditional mathematical practice, and is thus harder to read and understand (at least without training), it has other advantages that are important to the T1 model:

- In the postfix notation, everything happens in left-to-right source order. This participates to the no-magic policy.
- Since functions work over a shared data stack, they naturally receive several arguments and return several values without any extra syntax to that effect.
- As in Forth, "immediate" functions can be defined, that are invoked when encountered, in the middle of source code translation. This way, the source code itself can take over the interpretation process at any time and access the remaining of the source code in arbitrary ways. This allows defining new syntax on the fly, and, more generally, opens the way to generic powerful *metaprogramming*.
- Postfix source code can be readily serialized into an executable format running on *threaded code*, a well-known code generation method that can allow for a very small compiled code footprint.

Generic Object-Oriented Programming. In classic OOP (as in for instance Java or Go), functions can be attached to an object type and be invoked on an object instance (we then call them "methods"). Several functions may share the same name; the one which will be invoked will depend on the *runtime* type of the object on which the method is invoked (in C++ and C# terms, this is how virtual methods work). T1 goes one step further, in that the invoked function may depend on the runtime types of *all* arguments, not just the first one. Indeed, while Java, C# and other languages syntactically single out the first argument as "the instance on which the method is invoked", T1 considers all arguments on the same ground.

Only Dynamic Types. Consider the following Java code snippet:

```
class A {
    void foo(A a) {
        System.out.println("foo AA");
    }
    void foo(B b) {
        System.out.println("foo AB");
    }
}
class B extends A {
    void foo(A a) {
        System.out.println("foo BA");
    }
    void foo(B b) {
        System.out.println("foo BB");
    }
}
class C {
    public static void main(String[] args) {
        A x = new B();
        A y = new B();
        x.foo(y);
    }
}
```

This code will print “`foo BA`”. The variable `x` contains a reference to an object of type `B`, so the invoked method will be one of `B`, not one of `A`, even though `x` was declared as a variable of type `A`. On the other hand, `y` was also declared as a variable of type `A`, and this is the type which will be used to decide which of `B`’s `foo()` methods is invoked, even if the value which is then passed as parameter is really a reference to an instance of `B`.

This code snippet illustrates that Java uses two distinct notions of type for purposes of issuing calls to methods:

- For the first parameter, i.e. syntactically the instance “on which” the method is called, its *dynamic type* is used: this is the type of the instance, regardless of the apparent type of the expression that yields a reference to that object.
- For the other parameters (the ones within the parenthesized list), only the *static type* is used: this is the type syntactically attached to the expression, irrespective of the actual value at call time.

In T1, this duality is rejected; only dynamic types are used. This is part of the goal of OOP genericity: if all parameters to a function are treated on an equal basis, then they should all use the same kind of type analysis for purposes of method dispatch.

Use of only dynamic types does not mean that static typing cannot be performed; in fact, the T1 compiler is all about making thorough static type analysis. Rather, this means that the goal of static analysis is to work out the possible dynamic types of the parameters upon execution, and verify that there will indeed be methods matching each call. The *semantics* of the language are still defined in terms of the dynamic types of values, not of static types attached to expressions.

1.3 Memory Model

All *values* are references, i.e. pointers to object instances. This also holds, formally, for small integer types; e.g. a value of type `u32` (32-bit unsigned integer) that contains the number `5` is considered to be a pointer to an immutable instance that incarnates that number. In practice, for small integer types and booleans, these immutable instances don't actually exist in memory; formally, the booleans and small integers are still references.

There is no null pointer. When an object is created in memory, its fields are *uninitialized*, and reading an uninitialized field triggers a runtime error.

Tony Hoare introduced null references in ALGOL, mostly because it was easy to do so. He now calls this decision "his billion-dollar mistake". Null pointers imply the risk of null pointer dereference. In modern "big" systems, in which there is an active memory management unit, a null pointer dereference will reliably trigger a CPU exception, which the operating system will convert into some sort of interruption (e.g. a `SIGSEGV` signal on Unix-like systems). However, there can still be issues when the null pointer is taken as an array, with a large access index: the offset may make the access valid again, from the point of view of the MMU. On smaller systems without a MMU, null pointer dereferences cannot easily be trapped, leading to hard to debug errors.

Some languages do not have null pointers, in particular the Caml family. These languages demonstrate that avoidance of null pointers is possible and not especially hard, though it requires explicit initializers for all fields. T1 takes a "middle path" in which null pointers don't exist, but individual object fields may be uninitialized; this means that runtime checks will happen only upon field access, not on all pointer following actions. Static analysis might also be able to remove some of these checks.

There are no C#-style "value types". In C#, a value type is a `struct` that contains fields, and which is cloned when needed. The main reason to have value types is storage efficiency. Consider, for instance, an application that manages a large array of dates. In C#, this would use an array of the value type `DateTime`; all these instances would be concatenated in memory into a single allocated chunk. In Java, which does not have value types, an array of `Date` would be used, but this would really be an array of references to individually allocated `Date` instances. This would be likely to induce a much larger overhead for the memory allocator; it also increases access cost, since that is one extra layer of pointers to follow.

In order to recapture the storage efficiency of value types, T1 defines *embedding*: when an object type is described, or an array created, fields can be defined to be either values (i.e. references), or embedded sub-objects. An embedded sub-object is allocated within the encapsulating object, and there is no extra pointer. This changes the semantics (the embedded object is always there, and cannot be substituted for another), and thus is made explicit in the language; this is not an automatic optimization.

Since there are no value types, all parameters to functions are references, and all returned values are references as well. These values are exchanged over a common *stack*. This stack is separate from the in-memory structure that keeps track of function calls (in Forth terms, the data stack is not the system stack). In compiled code, thanks to the restrictions imposed by the compiler, the stack needs not be more than a transient abstraction; there is not necessarily a single dedicated memory area with a stack pointer.

Apart from the stack, functions may also declare local variables and locally allocated object instances (i.e. on the "system stack" in Forth terms). Local variables contain values, i.e. references to instances. Local variables are created when the function is entered, and destroyed when the function exits; notably, they are not bound to scopes smaller than a function body.

The Go language supports creating structures that can contain either embedded sub-objects, or pointers to other objects. The default (simplest) syntax in Go is for embedding, and Go supports value types in the C# sense. Since T1 core values are references, the syntax is different: structure types are by default references, and an extra syntax is used to make embeddings.

During interpretation, instances are allocated dynamically, and memory is managed with a garbage collector: unreachable objects are automatically reclaimed. Compiled code offers several options:

- The compilation phase may involve static allocation of instances which were created during interpretation, and are referenced from the produced code.
- Scope-based allocation is possible (i.e. "on the stack"). This is allowed by the compiler only insofar as it can statically determine, through escape analysis, that the object will not ever be used after the declaring scope has exited; moreover, for objects with a variable length (e.g. arrays), the actual length must be fixed at compile time. Such allocation does not necessarily happen on a physical stack.
- Dynamic allocation with automatic reclamation by the GC is possible. A point of T1, though, is to make such allocation optional (if the code does not use such allocation, the GC itself won't be included in the output).

All accesses to instances ultimately use special *accessor functions* which are automatically defined when the corresponding type is declared. When accessing array elements (by index), the accessor functions enforce strict bounds checking.

2 Lexing

We describe here how the T1 interpreter breaks down source code into individual tokens, upon which the T1 syntax is built. Since T1 is generically extensible (interpreted source code can, at any point, invoke itself and take over processing of the remaining of the source code), an arbitrary number of new parsing rules can be implemented by source code. The rules detailed below explain how parsing is done at the start of the source code processing.

> In Forth, the only lexing process is to aggregate sequences of non-space characters into “words”. All other syntax, e.g. literal strings or comments, is implemented by custom “immediate words”, which are functions that are invoked right away when encountered in the source stream. T1 implements a more complicated lexing process for ease of development.
>
> It shall be noted that Forth aims at offering support for development right on the target system, which may be embedded and constrained; this is one of the main reason for the very simple lexing process of Forth. In T1, the normal model is to make development on a dedicated powerful development workstation, distinct from the target system on which the code will run, which is why more expensive lexing is not an issue.

Input Characters. Source code consists in a number of text streams that are processed in due order. They are normally stored as individual files. Each stream contains bytes which are interpreted into *characters* using UTF-8 encoding; in this specification, a character is a Unicode “code point”, i.e. an integer in the **`U+0000`** to **`U+10FFFD`** range. Outside of literal strings, only ASCII characters (**`U+0000`** to **`U+007E`**) may appear.

> As will be described below, all the lexing really operates on bytes. In UTF-8 encoding, each ASCII character is encoded as a single byte with the same value, and all other code points are encoded as sequences of bytes of value **`0x80`** or more. The non-ASCII byte values that appear in literal strings can thus be simply copied, since, as we shall see, string values are really arrays of bytes. Moreover, the source itself can define and then invoke functions that can take over source code processing in arbitrary ways, and such functions may interpret source bytes differently. In that sense, it is not strictly necessary that source file uses UTF-8 encoding, only that the parts that rely on the lexing process described here use only ASCII characters outside of literal strings.
>
> However, it is expected that most source code writing will be done with text editors, that are likely to rely on, and enforce, a specific encoding charset. In the interest of maximum interoperability, it is here defined that all source code *shall* be UTF-8 encoded, and interpreters/compilers may enforce it.

Text breaks down into lines; each line is terminated by a newline character (**`U+000A`**). If a line ends with a CR+LF sequence (**`U+000D`** followed by **`U+000A`**), then this is considered to be equivalent to a single newline character.

Whitespace. *Whitespace* is any sequence of one or more characters in the `U+0000` to `U+0020` range, i.e. all ASCII control characters, and the ASCII space. Thus, tabulations (`U+0009`) and newline (`U+000A`) are whitespace. Whitespace characters separate tokens, but are otherwise not significant. Indentation, in particular, is a purely aesthetic choice with no impact on semantics. Take care that whitespace characters appearing within literal strings do not count as whitespace.

Comments. The character "`#`" starts a comment (unless it appears within a literal string or a character constant). The comment spans to the end of the current line, but does not include the newline character that terminates that line. If a comment appears on the last line of a file that does not end with a newline character, the comment spans to the end of the file. Comments are ignored; since, in general, a comment is immediately followed by a newline character, that newline character acts as whitespace.

Single-Character Tokens. Each of the following characters, when encountered outside of a literal string or character constant, counts as a token in its own right:

```
( ) [ ] { } '
```

Names and Numerical Constants. The lexer parses a *word* as a sequence of non-space printable ASCII characters (`U+0021` to `U+007E`), excluding the following characters:

```
( ) [ ] { } ' " #
```

The lexing process is "greedy": the longest sequence of allowed characters is assembled, and stops at the first disallowed character (from the list above), whitespace, or end-of-stream, whichever comes first.

Numerical Constants include the following:

- *Boolean constants* are the words "`true`" and "`false`".
- *Character constants* are all words that start with a backquote character ("`` ` ``", `U+0060`).
- *Number constants* are all the words that start with an ASCII digit ("`0`" to "`9`"), or a minus ("`-`") or plus sign ("`+`") followed by an ASCII digit.

Names are words which are not numerical constants.

Note that a word which starts with a sequence that introduces a numerical constant, but fails to parse as a valid numerical constant, triggers an error; it is not "demoted" to being a name.

In Forth, when a word is encountered, it is first evaluated as a function name; this works because Forth uses a strict define-before-use policy, so any word can be unambiguously matched against existing functions at this point. Only words which are not recognized as function names will be re-interpreted as possible numerical constants. A side effect is that it allows defining functions with names such as **`42`**, a feature which is more confusing than useful.

In T1, we allow referencing functions and types that will be defined later on, and thus we cannot use numerical interpretation as a fallback for unknown function names. The non-reinterpretation of invalid numerical constants as names is meant to promote readability: looking at the start of a word is enough to know whether it is a numerical constant or a name; it also allows later versions of T1 to enrich the syntax with more numerical constants, e.g. floating-point values, without breaking backward compatibility.

A consequence is that function names cannot start with a digit, such as Forth's "**`2DUP`**".

Number Constants. Valid number constants are:

- a sequence of ASCII digits, interpreted as an integer value in base 10;
- the sequence "**`0x`**" or "**`0X`**", followed by one or more hexadecimal digits (hexadecimal digits are ASCII digits "**`0`**" to "**`9`**", ASCII uppercase letters "**`A`**" to "**`F`**", and ASCII lowercase letters "**`a`**" to "**`f`**"), interpreted as an integer value in base 16 (case is not significant);
- the sequence "**`0b`**" or "**`0B`**", followed by one or more binary digits ("**`0`**" or "**`1`**"), interpreted as an integer value in base 2;
- any of the above, preceded by a minus sign ("**`-`**") or a plus sign ("**`+`**"); the minus sign makes the value negative, while the plus sign does not change the value and is purely cosmetic;
- any of the above, followed by a suffix in the following list, and defining the constant to have the corresponding modular integer type: **`i8 i16 i32 i64 u8 u16 u32 u64`**

If the number constant does not have an explicit type suffix, then it has plain integer type (**`std::int`**, as will be defined in section 4). If the value does not fit in the allowed range for the target type, then an error is raised.

Character Constants. A *character constant* describes an integer value of type **`std::u8`** in the 0 to 126 range (inclusive). Valid character constants consist in a backquote character (**`U+0060`**) followed by:

- a single ASCII character in the **`U+0021`** to **`U+007E`** range, excluding the backslash character ("**`\`**");
- an escape sequence that starts with a backslash, followed by one character:

- `\s` stands for space (`U+0020`);
- `\t` stands for tabulation (`U+0009`);
- `\r` stands for carriage return (`U+000D`);
- `\n` stands for newline (`U+000A`);
- `\'` stands for quote (`U+0027`);
- ``\` `` stands for backquote (`U+0060`);
- `\"` stands for double-quote (`U+0022`);
- `\\` stands for backslash (`U+005C`).

In all cases, the character constant stands for the numerical value that corresponds to the represented code point.

> Character constants are deliberately limited to plain ASCII because they have type `std::u8`, following the decision that "normal" strings really are sequences of bytes. This will be explained in more details in section 4.

Since character constants are self-terminated (inspection of their contents is enough to decide that no extra character follows in the token), they need not be separated by whitespace from the next token. Thus, "`` `ab ``" is parsed as two tokens, the character constant for lowercase letter "a", then the one-character name "`b`". This is of course confusing, so don't do that.

Literal Strings. A *literal string* represents a value which is a sequence of bytes. Such a token starts with a double-quote character "`"`" and ends at the next unescaped double-quote character. The following rules apply:

- The starting and ending double-quote characters are not part of the string contents.
- Bytes appearing in the string literal, other than backslash and newline, are part of the string literal. This includes all byte values, even ASCII control characters (note that a CR+LF sequence at the end of a source text line counts as a single newline character, which cannot appear unescaped within a literal string).
- When a backslash appears within a literal string:
  - If the backslash is immediately followed by the newline (or CR+LF) that ends the line, then this is a *line escape*: the next line must start with zero or more whitespace characters (except newline), followed by a double-quote character; the backslash, newline, whitespace and double-quote character are then skipped, and parsing of the literal string continues after the double-quote character.

- Otherwise, the backslash character must begin an escape sequence. Escape sequences are:
  * escape sequences that may appear in character constants;
  * `\x` followed by exactly two hexadecimal digits, standing for the byte whose value is expressed in hexadecimal by these two digits;
  * `\u` followed by exactly four hexadecimal digits, standing for the UTF-8 encoding of the code point whose value is expressed in hexadecimal by these four digits;
  * `\U` followed by exactly six hexadecimal digits, standing for the UTF-8 encoding of the code point whose value is expressed in hexadecimal by these six digits.

  Note that hexadecimal digits are ASCII digits "`0`" to "`9`", uppercase letters "`A`" to "`F`", and lowercase letters "`a`" to "`f`". Case is not significant for hexadecimal digits. Unrecognized escape sequences trigger errors.

For instance, this literal string:

```
"Hello\
" World!"
```

uses a line escape and has contents "Hello World!".

The four following strings:

```
"café"
"caf\xc3\xa9"
"caf\u00E9"
"caf\U0000E9"
```

all define the same sequence of five bytes. Take care that "`\x`" escapes allow inclusion of arbitrary byte values which do not necessarily correspond to the valid UTF-8 encoding of a sequence of code points.

Apart from line escapes, newline characters may not appear into a literal string (but a "`\n`" escape sequence can be used to include a newline character in the string contents). Thus, a literal string may span several lines only if each line (except the last) ends with a line escape. Moreover, a string literal must be terminated before the end of the current text stream; string literals do not span across files.

Since the whitespace characters that are part of a line escape do not include the newline character, a comment is not possible within that whitespace.

3 Names

In T1, functions, types and local variables have names. A name is a sequence of characters (unicode code points). In general, any sequence of code points is usable for any purpose. However, names that are encountered within source code are subject to some processing which modifies their interpretation and restricts their syntax.

The lexer parses a name as a token that contains only printable non-space ASCII characters, excluding a few "forbidden" characters (parentheses, braces...). Such a name may be "qualified" or "unqualified":

- A *qualified name* contains a single instance of the sequence "::". The part before the sequence is the *namespace*, and the part after is the *raw name*.[1]
- An *unqualified name* does not contain the sequence "::". Thus, raw names (obtained from removing the namespace from a qualified name) are unqualified.

The general model of source code interpretation is that qualified names designate a specific entity, and unqualified names are interpreted depending only on the syntactic context. For instance, under normal conditions, the interpreter reads the next token and expects it to be either a numerical constant, a literal string, or a function name. In this last case:

- If the name is qualified, then this designates exactly that function.
- If the name is not qualified, then it will be matched against the following, in due order:
  - If the name matches an accessor for a locally allocated variable or instance, then the name is interpreted as an invocation of that accessor.
  - If the name is part of one of the currently defined aliases (imports), then it designates the function that the alias maps to.
  - Otherwise, the name is considered to use the currently active namespace.

Namespaces are used to keep track of defined functions and types, and to avoid spurious collisions. Normal source code should contain very few qualified names:

- At any point in the source code, there is an active namespace in which new functions and types are defined.
- Access to names in other namespaces is normally done through aliases and import lists.

[1] In general a name shall not contain more than one instance of "::". If it does, then all operations that split the name into a namespace and a "raw" name use the first (leftmost) occurrence of "::" as splitting point.

- The developers are supposed to have full knowledge of "their" namespaces, and thus avoid internal collisions.

There are no visibility rules, i.e. public and non-public functions and types. Every such element *can* be accessed by using a qualified name. However, good software engineering practice is to refrain from doing so in the general case. Functions and types can be added to the *export list* of the currently active namespace: such an export list can be imported from other namespaces with an "`import`" clause, which locally defines corresponding aliases for the exported names. In that sense, "public" functions can be defined by making them part of the export list of their namespace, which documents the intent of making them callable from other namespaces.

More details on import lists and aliases are given in section 6.8.

All namespaces that start with "`std`" are reserved for the T1 implementation. Notably, the export list for "`std`" itself is automatically imported, and it defines aliases for all the core syntactic constructions and types. Source code can, at any time, clear the current list of aliases, including those from the "`std`" export list.

4 Types

All T1 values are *references* to *instances*. This includes the basic types (booleans, small integers...). Formally, a value for the integer "5" is considered to be a reference to an immutable object instance that represents that integer; such instances are virtual and cannot be really created in memory. This definition allows us to define the T1 type system without making special cases for such basic types. In practice, there are some restrictions in compiled code that avoid type punning that would be expensive to implement.

4.1 Sub-Typing

Sub-typing is a mechanism which incarnates promises of functionality. When type **`bar`** is a sub-type of **`foo`**, then it means that whenever a function expects as argument a reference to an object of type **`foo`**, it may receive instead a reference to an object of type **`bar`**, and things "should work". As we shall see later on, the only thing that can be made with values is to call functions on them, and function calls are dynamically mapped based on the runtime types of their arguments; thus, making **`bar`** a sub-type of **`foo`** means that for every function that takes as input a **`foo`**, a function of the same name that accepts a **`bar`** is defined.

Whether such promises are fulfilled or not does not impact sub-typing. During interpretation, an unfulfilled promise will trigger a runtime error at the time the function is called. The compiler statically checks that such a situation does not occur; in that sense, the compiler does not check that there are methods for **`bar`** that correspond to all methods for **`foo`**, only that there are such methods for all calls that can actually occur in the compiled code.

Sub-typing has the following rules:

- Every type is considered to be a sub-type of itself. A *strict sub-type* of type "**`T`**" is a sub-type of "**`T`**" that is not "**`T`**" itself.
- All types are sub-types of "**`std::object`**".
- Strict sub-typing is an acyclic graph. A given type is a sub-type of itself, but shall not be a sub-type of any strict sub-type of itself.
- Sub-typing rules can be added to any type at any time, provided that they don't create cycles. Basic types are an exception, in that they cannot be sub-typed, or made strict sub-types of any other types except "**`std::object`**". Note that sub-typing can be added even on types for which instances have already been created.

Sub-typing is distinct from both embedding and extensions, which will be covered later on.

If **`A`** is a sub-type of **`B`**, then **`B`** is a *super-type* of **`A`**. A type can have several direct sub-types and several direct super-types. Sub-types and super-types are not ordered.

Sub-typing is somewhat similar to interfaces in Java and C#: a mechanism to tag types, and promise existence of methods, without actually implementing them. However, in both Java and C#, such promises are checked by the compiler: a class that declares that it implements a given interface, but fails to provide the relevant methods, will be rejected at compilation. This does not occur in T1, which concentrates on actual usage: a compilation error is triggered not by failure of providing a method that matches a given sub-typing relationship, but by trying to call such a method.

4.2 Basic Types

Basic types are the following:

- **`std::object`** is the root of the sub-typing graph. Instances of **`std::object`** cannot be created. The basic equality and inequality functions ("**`=`**" and "**`<>`**") are defined on **`std::object`** and implement (in)equality of references. Due to the way function lookups are performed, this behaviour is inherited by all types, unless explicitly overridden.
- **`std::bool`** is the boolean type. The two possible values are "**`true`**" and "**`false`**".
- **`std::int`** is the default integer type. It has a range which depends on the current architecture and implementation, but which is large enough to serve as index value in arrays. It is signed: its range is $-m$ to $m-1$ for a given integer $m = 2^l$. Operations on **`std::int`** are checked; any overflow condition triggers a runtime error.
- **`std::u8`**, **`std::u16`**, **`std::u32`** and **`std::u64`** implement unsigned integers modulo 2^8, 2^{16}, 2^{32} and 2^{64}, respectively. Since they are modular integers, they have "wrap-around" semantics, and never overflow.
- **`std::i8`**, **`std::i16`**, **`std::i32`** and **`std::i64`** are the signed types corresponding to the unsigned modular types. For instance, **`std::i8`** is for integers in the -128 to 127 range. They implement wrap-around semantics, just like unsigned types; like Java and unlike C, computations on signed types that exceed the range yield well-defined results.

There are no automatic conversions. Contrary to C, no value can be used as a boolean, except the values of type **`std::bool`**. Arithmetic operations such as addition cannot involve distinct types; e.g. you cannot add a **`u8`** to a **`u16`**. Conversion functions are provided. In C and C++, many automatic conversions are applied; in Java and C#, only widening conversions (that conserve the mathematical value) are implicit. In T1, all conversions must be explicit: this is meant to force the developer to have a clear mental picture of what happens to the data.

For the default integer type, typically used for indexing into arrays, the following language designs (at least) are possible:

- Use a native type of the architecture, with wrap-around semantics. This is what Rust does with the `isize` and `usize` types. In C, there is an unsigned integer type which is what `sizeof` returns and `memcpy()` expects; the standard headers give it the name "`size_t`".

- Use a type with a defined, fixed width, typically 32 bits. This is the Java road, with `int`. This implies some limitations when machine memory sizes have grown so much that indexes beyond 2^{31} are no longer ridiculous. In C#, some extensive contorsions were made to allow indexing with both `int` and `long`.

- Transparently expand integers into *big integers* with unlimited range, bounded only by the RAM required to represent such values. Python uses this strategy; it is also encountered in many Scheme implementations. Computations on small integer values can be done without dynamic allocation, but large integer values incur some performance loss.

- Use a type with a defined range (that may depend on the architecture) but detect overflows and transform them into actual errors. This is reminiscent of Ada.

- Make something weird like JavaScript: there are no integers, only floating point values. When going out of range, values become approximate.

T1 follows the Ada way for several reasons:

- Wrap-around semantics, or the C "undefined behaviour" when exceeding range on a signed type, make for devious bugs which are often security issues. Secure code must often include extensive analysis and explicit checks to make sure that overflows don't occur; it is safer to make all checks by default, and possibly suppress them when the compiler can be convinced that no overflow occurs. We may say that integer overflow checks are needed in the same way as bounds checking is needed for array accesses.

- A fixed-width type is too limiting; e.g. a 32-bit type is too expensive for small 16-bit microcontrollers, and yet not large enough for large 64-bit systems.

- Big integers are seductive but incur some extra costs; in particular, some form of dynamic memory allocation is needed, and this goes contrary to the strict RAM discipline and memory safety that T1 strives to achieve. Moreover, big integers cannot be implemented in constant-time.

T1 implementations may use a slightly smaller range than expected. For instance, on a 32-bit architecture, `std::int` values may have a range limited to -2^{30} to $2^{30} - 1$, i.e. 31 bits with signed interpretation, not 32 bits. This is done in order to allow an in-memory representation that is compatible with pointers: pointers to instances are normally aligned, hence use even 32-bit values; thus, integers can be represented as odd values, the least significant bit being used to mark the value as an integer. That kind of trick is common in Scheme and OCaml implementations; it allows storing integers in pointer fields in a way which works well with runtime type checks and garbage collection.

4.3 Strings

There is no dedicated "character string" type. Strings are arrays of bytes. Literal strings define statically allocated arrays of bytes. The name "`string`" is provided as an alias for the name of the type for an array of bytes (which is "`(std::u8 std::array)`", as will be described later on).

In C, there are no real character strings, only arrays of `char` with a terminating zero. T1 does not need or use a terminating zero, because its arrays have a definite length accessible at runtime.

In the infancy of computers, "characters" were believed to be simple, atomic elements that could be represented with a simple, small, fixed-width type, e.g. "`char`" in the C language. It soon appeared that different languages required more characters, and "code pages" were invented to incarnate the interpretation of bytes into characters. Code pages implied huge interoperability issues, and the problems were made much worse when non-alphabetic scripts such as Japanese had to be taken into account.

Unicode is a unifying effort that tries to remove the need for code pages. Unicode defines *code points*; initially, each code point was a 16-bit integer, but this proved too limited, and code points can now use up to 22 bits (valid code point values are in the 0 to 0x10FFFD range). Java defined its `String` type to be a sequence of `char` values, a 16-bit type, as per the first version of Unicode. A more modern redefinition would use a 32-bit integer type, so as to represent the whole range of possible code points.

This is, however, an illusion. Unicode defines code points, not "characters". Consider for instance the English word "café" (an import from French, but still a valid English word); the last character ("é") admits two representations in Unicode. The first one is a single code point `U+00E9` ("latin small letter E with acute"); the other one is a sequence of two code points, `U+0061` ("latin small letter E") followed by `U+0301` ("combining acute accent"). Thus, a single "character" may consist of several code points. A lot of combinations are possible (in particular in Hangul, the Korean writing system) and it would not be practical to map all of them into a single numerical type. Therefore, any sufficiently advanced Unicode-aware processing of text must be able to accomodate variable-length representations of characters.

A simpler model is represented by Go: strings are just bytes. When the bytes must be parsed as text, then they are decoded as per UTF-8 rules. UTF-8 has some nice properties:

- Every ASCII character is encoded as a single byte whose value matches the ASCII code of the character. For instance, the ASCII code of "e" is 0x61, and it is encoded as a single byte of value 0x61. Thus, ASCII is "preserved" by UTF-8 encoding.
- All other code points are encoded as sequences of bytes with values no less than 0x80; none of these bytes may be misinterpreted as an ASCII character.

Using UTF-8 means that technical processing, in particular for text-based protocols such as HTTP, can use the traditional one-character-per-byte model, provided that the processing uses only ASCII characters, and bytes with values 0x80 or more are just kept together. This has the additional benefit of not *enforcing* UTF-8 encoding; this is handy when, for instance, exploring file directories, where file names are OS-provided sequences of bytes which need not be valid UTF-8.

T1 follows the Go way and uses byte arrays for strings.

Functions that use and return strings assume immutability: a string instance should never change once initialized. However, T1 does not enforce this property: when an array of bytes is used as a string, it is up to the developer to refrain from modifying its contents as long as it is used elsewhere with immutability semantics.

The T1 *compiler* enforces immutability of all statically allocated instances, and this includes the instances corresponding to literal strings.

Enforced immutability would make programming "safer" at the expense of extra allocations. For instance, if bytes are read from a network interface, then these bytes are written into a buffer. To interpret that buffer as an immutable string, there are several options:

- Copy the bytes into a newly allocated read-only object. This is what is done in Go when a "`[]byte`" value is cast into a "`string`". Such a mechanism requires dynamic allocation.
- "Lock" the buffer with a flag checked at runtime for each write access. This requires room for that extra flag, and, indeed, runtime checks, which may have a non-negligible cost. Unlocking would have to be performed as well. Also, any error will be reported only at runtime, which is undesirable in general (compile-time error reporting is much preferred).
- Use complex borrowing semantics to ensure that concurrent modifications don't occur. This is what Rust does, with far-reaching consequences on the application structure (what is deemed by the colloquial euphemism "fighting the borrower").
- Don't do anything; just *document* that a given string, when provided to a function, may be retained and used after that function has returned, and therefore must not be modified.

T1 follows the last of these options, based on my own development experience: I don't tend to make bugs related to immutability confusion, and thus the enforced extra safety does not seem to be worth the extra costs for that property. This is a personal judgement call, and I might add a truly immutable string type in a later version.

4.4 Structures

New types are built as *structures*. A structure contains *fields* and *embedded sub-structures*. A field contains a value (i.e. a reference); an embedded sub-structure is an instance of another type, which is created along with the encapsulating instance. Though implementations may vary, the intended effect is that fields appear in the memory layout in the order they appear in the structure, and embedded sub-structures are really embedded, i.e. use for their own memory layout the corresponding chunk of memory of the encapsulating structure.

Each field has a type, which is a filter on possible values of the field: these values must have types which are sub-types of the field type. This is a side-effect; the primary function of the field type is to qualify the declaration of accessor functions for that type. Embedded structures also have a type, which defines which structure is embedded. Only other structures may be embedded; it is not possible to embed basic types (and it would not make much sense either). Embedding is acyclic: a structure may not directly or indirectly embed itself.

Arrays of fields and arrays of embedded structures can be defined, with a fixed number of elements.

Consider the following example. We suppose that the current namespace is "`def`", and that the "`bar`" type is defined, or to-be-defined, in that namespace.

```
struct foo
    x int
    b1 bar
    b2 && bar
    c1 16 bar
    c2 && 16 bar
end
```

A structure named `def::foo` is defined, with the following contents:

- A field called `def::x`, that may contain values of type `std::int` or sub-types thereof (but there cannot be strict sub-types of `std::int`). Note that the unqualified name "`int`" is converted by an active alias to "`std::int`", because that alias is part of the export list from the namespace `std`, which is imported by default.
- A field called `def::b1` for references to instances of `def::bar` (or sub-types thereof).
- An embedded structure of type `def::bar`, with name `def::b2`.
- An embedded array of 16 references of type `def::bar`, with name `def::c1`.
- An embedded array of 16 embedded sub-structures of type `def::bar`, with name `def::c2`.

No two elements of a structure may have the same name, regardless of their respective kinds.

4.4.1 Closing

When first encountered, a structure type is created in an "open" state. This means that its name becomes known, but the full contents are not yet defined. As long as a structure is open, new fields and embedded elements (sub-structures, arrays, and arrays of embedded sub-structures) can be added to the structure type. Once the type is *closed*, no new contents may be added. In the example above, `def::foo` is still open: new fields and embedded elements could still be added to the structure. The "`end`" keyword does not close the structure; it merely exits the syntactic construction that is used to add elements to a structure.

Similarly, the `def::bar` structure, if not yet defined at this point of the source code, has been automatically created, in open state and with no initial contents, when the name "`def::bar`" was first encountered (i.e. when the field `def::b1` was defined).[2]

A structure will be closed in the following circumstances:

- When closed explicitly with a specific function call on the type instance (i.e. the instance of type "`std::type`" that represents this type).
- When an instance of the structure is created. Instance creation implies memory allocation, which needs the layout to be fixed.
- When an encapsulating structure is closed. For a structure to be closed, all the structures it embeds must first be closed. Since the embedding relationship is acyclic, this process converges.
- When the structure is made part of code being compiled.

Conversely, sub-types and super-types can be added to a closed structure; that is, even after `def::foo` is closed, new structures can be defined and made sub-types of `def::foo`, and `def::foo` itself can be made sub-types of other structures; these additions are immediately inherited by existing instances of `def::foo`.

4.4.2 Instantiation

When a structure type `T` is defined, a function with the same name is created. That function takes no parameter, and returns an instance of `std::type` which represents the type `T`.

A dedicated function `std::new` takes as parameter a `std::type` instance, and creates a new instance of the represented type. The fields of the new instance are set to uninitialized state (except fields of boolean or modular integer types, which are set to their default `false` or zero values); this also applies, recursively, to all embedded elements.

Calling `std::new` on `std::type` instances that do not represent structure types triggers an exception.

[2]Strictly speaking, the `def::bar` structure, if created at this point, is in *implicit* open state; if it was not explicitly defined at the time `def::foo` is closed, then an error is reported, under the assumption that a type which was never explicitly defined anywhere is probably a typing mistake.

4.4.3 Accessors

Structure contents can only be inspected and altered through dedicated *accessor functions*. These special functions are created when the structure is closed. The accessors use the element names, depending on the kind of element:

- For a field of name `def::x`, the functions `def::x` and `def::->x` are defined, to read and write values to the field `def::x` of an instance, respectively. The accessor `def::Z->x` clears the field, i.e. sets it to uninitialized state. The accessor `def::x?` tests whether the field is initialized or not.
- For an embedded structure of name `def::x`, one accessor function of name `def::x&` is defined, which takes as input a reference to an instance of the encapsulating structure, and returns a reference to the instance embedded within it.
- For an array of references, with name `def::x`, the functions `def::x@` and `def::->x@` read and write values into the array slot indexed by a given `std::int` value. Also, `def::Z->x@` clears a slot, and `def::x@?` tests its initialization status. Finally, `def::x*` initializes an array instance (of the right type) to provide an array view of the references.
- For an array of embedded sub-structures, with name `def::x`, the `def::x@&` accessor returns a reference to one of the embedded sub-structures, by `std::int` index, and `def::x*` initializes an array view of the sub-structures.

Fields, and slots in embedded arrays of references, are initially uninitialized. There is no null value; reading an uninitialized field triggers a runtime error.

For booleans and modular integers, i.e. the `std::iXX` and `std::uXX` types, corresponding fields are always initialized. Their starting value is `false` (for booleans) or zero (for modular integers), and the clearing accessors restore that value. The test accessors (`def::x?`, `def::x@?`) then always return `true`. Note that plain `std::int` fields are not in this situation, and can be truly uninitialized.

4.4.4 Extension

A structure may *extend* another structure; this is a combination of sub-typing and embedding. When structure `B` extends the structure `A`:

- `B` is defined to be a sub-type of `A`.
- `B` embeds an instance of `A`, under the name of `A` (that is, the accessor `A&` is defined, that takes as input parameter a reference to an instance of `B`, and returns a reference to the embedded instance of `A`).
- Accessors for elements of `A` can be used on an instance of `B`, and will access the corresponding elements in the instance of `A` which is embedded in `B`.

Since extension is both embedding and sub-typing, it combines the requirements of both; notably, extension cannot be done in a closed structure, and the extension relationship is acyclic.

A given structure `B` may directly extend a given structure `A` only once (this is implied by the fact that the extended structure is embedded under its own name, and names are unique within structure contents). However, a structure may extend several other structures. This is a multiple inheritance model, which is powerful but implies some ambiguous situations. Suppose, for instance, that:

- Structure `A` has a field named `x`.
- Structure `B` extends `A`.
- Structure `C` extends `A`.
- Structure `D` extends `B` and `C`.

In that situation, the `x` function, which is the read accessor for the field of the same name in `A`, could be invoked on an instance of `B` and on an instance of `C`. Since `D` extends `B`, the accessors that accept an instance of `B` will also work on an instance of `D`. However, the same can be said about the accessors that accept an instance of `C`. In fact, since `D` embeds both a `B` and a `C`, and each embeds an `A`, the structure `D` indirectly embeds *two* instances of `A`, and it is unclear which one is supposed to be used when reading the field `x`. Therefore, invoking `x` on an instance of `D` triggers an error.[3]

It is possible to make an explicit decision, by defining a function called `x`, attached to type `D`, which then selects the instance to use:

```
: x (D) B& x ;
```

This snippet reads as follows:

- The "`:`" token starts the definition of a new function. It is followed by the name of that function (`x`).
- The "`(D)`" expression registers the new function to the type "`D`"; that is, if a function call for name `x` is encountered, and at that time the top element on the stack has type `D`, then this function shall be called (and not, in particular, the accessor function which is registered on type `A`).
- Afterwards follows the function body, which here consists in two successive function calls: `B&`, which returns a reference to the sub-structure embedded in `D` under the name `B`, and then `x`. Since that `x` call will operate on the `B` instance returned by `B&`, it will use the `A` instance embedded in that `B` instance, and not the one embedded in the `C` instance which is also embedded in `D`.

[3] As we shall see, part of the work of the compiler is to prove that such an error cannot happen in a given piece of code.

- The semicolon token (“`;`”) terminates the function body.

Thus, this new function explicitly chooses `B`, not `C`. Since it is registered with type `D`, which is a sub-type of `A`, it will have precedence over the `x` accessor function defined on `A`, when invoked over an instance of `D`.

Type extension in T1 maps to the Java extension of classes, while sub-typing corresponds to the Java extension of interfaces. Historically, Java had only classes, and interfaces were added afterwards to compensate for the lack of multiple inheritance. The Java inheritance has several facets:

- Inheritance of storage: state held in the superclass is also contained in the subclass instance. In T1, this is done with extension.
- Inheritance of behaviour: methods attached on the superclass also work with subclasses. This is what sub-typing provides in T1.

4.5 Arrays

Array types are defined on-demand. For a given type `T`, the type “array of `T`” has the name:

```
(T std::array)
```

(including the parentheses). This name is not a name token, as returned by the lexer; however, as will be explained in the description of the interpreter, the array type name mimics a sequence of code that, when processed by the interpreter, yields a reference to the `std::type` instance that represents the array type. In fact, the `std::array` function *creates* the array type if it does not already exist, and registers all accessor functions for array instances.

Array instances really are *views* on a chunk of memory. An instance of the array type, when instantiated but not initialized, points to nothing, and calling data accessors triggers an exception. An array instance is populated in basically four ways:

- Make the array instance point to a sequence of values or embedded structures within a given structure instance. If the sequence of values or embedded structures was declared with the name `def::x`, then this is done with the `def::x*` accessor function.
- Make the array instance point to a locally allocated sequence of values or embedded structures. For a local name `x`, this is done by using the `x*` pseudo-function name.
- Dynamically allocate a new memory chunk, with a specified length. When dynamic memory allocation is supported, this is done with the `std::make` function.

- Initialize the array instance as a sub-array of another array instance. The sub-array must be entirely contained within the source array. This is done with the **`std::sub`** function. An array instance can be reinitialized as a sub-array of itself with **`std::subself`** (which is just a shorthand for **`std::sub`** with the instance used for both operands).

There is thus always an indirection layer when accessing memory chunks. Memory chunks themselves are not objects, i.e. they cannot be accessed directly, and do not have a T1 type. Native code called from T1 can obtain a direct pointer to the data, subject to some caveats (in particular, objects may be moved in memory by the garbage collector, if used; and locally allocated objects cease to exist when the owner function returns): T1 memory safety guarantees that all array accesses are "safe" (e.g. out-of-bounds accesses trigger a runtime error, and all reachable objects are maintained in memory to avoid dangling pointers), but native code can bypass such safety features.

> The Go language has *arrays* and *slices*. An array is a sequence of value, and a slice is a view on such a sequence. Most operations that work on arrays also work on slices. In T1, the Go arrays become "chunks of memory" and are not directly accessible; the T1 "arrays" are equivalent to the Go slices.
>
> Thus, a T1 array type can be thought of as a structure with three fields: pointer to the actual object that contains the data, offset and length of the chunk within that hidden object. There is no notion of "capacity" as in Go (T1 arrays are not intrinsically growable).

An "array of **`T`**" is an array of references (to elements of type **`T`**, or sub-types thereof). A newly created memory chunk will have all slots uninitialized (or set to **`false`** or zero, for booleans and modular integers). The following accessor functions are defined:

- **`std::make`** dynamically allocates a new memory chunk, and initializes the array instance to point to that chunk.
- **`std::sub`** initializes an array as a view of a chunk of another array. The source array must have been initialized.
- **`std::subself`** merely duplicates the argument, then calls **`std::sub`**.
- **`std::init?`** returns **`true`** if the array instance was initialized, **`false`** otherwise. If the array instance was not initialized, then calls to the other functions below will trigger a runtime error.
- **`std::length`** returns the length of the array (number of elements).
- **`std::@`** and **`std::->@`** read and write a value from an array slot or to an array slot, respectively, indexed by an **`std::int`** value. Array indexes start at zero.
- **`std::Z->@`** clears an array slot, and **`std::@?`** returns its initialization status. For an array of booleans or modular integers, clearing a slot means setting it to **`false`** or zero, and **`std::@?`** always returns **`true`**.

Types for arrays of embedded structures can also be obtained with **`std::array&`**. The expression:

```
(T std::array&)
```

will return an array type that, when instantiated and initialized, is a view to a sequence of contiguously allocated instances of **`T`**. For such an array, the accessors that use or return references (**`std::@`**, **`std::->@`**, **`std::Z->@`** and **`std::@?`**) are not defined; instead, the accessor **`std::@&`** returns a reference to one of the structures embedded in the array.

Any other type can present an array-like interface by defining the appropriate methods. It shall be noted that support for the **`std::sub`** function implies that any array-like type must be able to present some of its contents in a contiguous sequence in memory.

> The main idea behind arrays-as-views is to make it so that any function that can work on arrays will also work on a sub-array. In practical Java or C# code that processes binary data (e.g. I/O code), several methods are often needed, e.g. a **`write()`** that takes as parameter an array of bytes, and another **`write()`** method that takes as parameters an array of bytes, a start offset and a length. Array views are meant to avoid these multiple methods. Moreover, they allow user-defined types with array-like interfaces to be used instead (e.g. a growable vector of bytes).

4.6 Generics

The on-demand creation of array types is an example of how generic types are managed in T1. In full generality, container types are meant to be created with metaprogramming. For instance, growable array types are created with **`std::list`**. The following sequence of code:

```
(u8 list)
```

will return an **`std::type`** instance that represents growable arrays of bytes. This is a normal structure type (albeit with a name that is not a qualified name token), and the **`new`** function on that type instance will return a new growable array of bytes with an initial size of zero. Each growable array type is a sub-type of the corresponding array type, and offers the relevant accessor functions, as well as some extra functions to append or remove elements.

Syntactic facilities are made available to users, in order to define their own generic types. This is not restricted to making new types parametrized by other types; this is more an expression that source code can define functions and invoke them during the interpretation itself, to process further source code and define other functions in arbitrary ways.

In Java, an important point of generics is that they impact the type analysis, but do not create new types: **`ArrayList<String>`** and **`ArrayList<Date>`** both use the same **`Class`** instance (their respective **`getClass()`** methods return the same object), and cannot be distinguished from each other at runtime. This is an historical consequence of generics being added only relatively late in the language (for Java 5). The generics are handled as an extra layer at compile-time that is used to avoid having the developer make explicit casts and risk the dreaded **`ClassCastException`**.

Conversely, in C#, **`List<string>`** and **`List<DateTime>`** are distinct types with distinct runtime **`Type`** instances (as returned by **`GetType()`**). The C# compiler analyzes the source code to make sure that, when replacing the type parameters with actual types (that comply with the expressed restrictions on the type parameters), the resulting code will still be valid; but the runtime machine will create as many distinct types as necessary. T1 works similary to C#, minus the initial abstract analysis: in T1, we do not really care whether things *would* work with some large categories of types, but whether they *will* work with the types that the source code actually uses.

5 Functions

Every piece of code in T1 is a *function*. A function has a name (which is a character string) and is *registered*; the registration is what makes the function callable. Several functions may have the same name, but will then differ by the types under which they are registered.

5.1 Runtime Model

Function parameters, and returned values, are exchanged on a *stack*. The stack contains only values, which are references. Every function extracts the parameters it needs from the stack, and pushes back its returned values on the stack. This inherently allows functions to return several values.

A function, when invoked, has an *activation context*, which is traditionally called a *stack frame*. This is disjoint from the stack described above. Implementations may use a stack structure to allocate activation contexts; in that case, that stack structure is often called the "system stack", while the normal stack for values is called the "data stack". Here, we will strive the avoid the confusion by using the expression "activation context", and reserving the term "stack" for the data stack.

The activation context is a transient memory area that will contain the local variables for the function, and may save the current execution point for the function. When a function calls another function, the current instruction pointer is saved in the function activation context, and a new activation context is created for the called function; when that called function returns, its activation context is released, and the instruction pointer is restored from the activation context of the caller. Exactly how this happens is an implementation detail.

Local variables are slots that can receive values; during translation of the source code, local variables have names, which allows source code to issue read and write instructions for these variables.

> In Forth, the system stack is explicit, with words `R>` and `>R` to move values from the system to the data stack, and back. Since some tasks may require more complicated data movements (the usual example is adding tridimensional vectors together), Forth also defines local variables, which are usually located on the system stack. Local variable names are translated at compilation time into depths on the system stack, which means that local variables don't interact well, or at all, with facilities that access the system stack, such as explicit words (`R>` and its ilk...), or loop counters. Thus, a function may use the system stack explicitly, *or* use local variables, but should not try to mix both.
>
> For T1, which is not encumbered by compatibility with existing legacy code, it seems simpler to avoid the complications and normalize on a single system. Thus, local variables have been chosen, and the system stack is not made visible to user code, except as the "activation context" abstraction.

Locally allocated instances are an extension of local variables: these are object instances that are part of the activation context. This corresponds to automatic variables in C or C++; arrays of references or embedded structures can be obtained that way. These locally allocated instances are nominally destroyed when the owner function exits. T1 does not have destructors (in the C++ sense) or finalizers (in the Java sense), thus the notion of "destruction" really means memory deallocation. During interpretation, the garbage collector is used for such instances, meaning that local allocation is not different from normal heap allocation. In compiled code, allocation is really done in the activation context, and has some restrictions so that memory safety is maintained in all its facets (notably guaranteed maximum stack growth): the size of such instances must be known at compile-time, and instances shall not "escape" to outer contexts, i.e. remain reachable once the owner function has returned.

> A typical use for stack allocation is creation of an array view instance to designate a chunk of an array provided by a calling function, for purposes of using that new array view instance as parameter to another nested function. Such operations should be doable even when compiling for targets that do not support dynamic memory allocation. Another use is assembly of a small character string, e.g. for immediate display.

5.2 Function Invocation

The only way to invoke a function is by name. Function names may be arbitrary; syntactically, a name token is used, and unqualified names are translated to qualified names by the parser, thus most function calls should use qualified names.

To be callable, a function must be *registered*. A function registration includes its name, and parameter types. For instance, this code defines and registers a function:

```
: foo (int string)
    # Here goes the function body
```

If the current namespace is `def`, then the function is registered under the name `def::foo` and with two parameter types, `std::int` and `std::string`. The intent is that if some code calls the function "`def::foo`", and at that exact time, the runtime types of the top two stack elements are `std::int` and `std:string`, respectively, then the function defined above shall be the one to be called. Types are provided in "stack order", i.e. the rightmost element is the top-of-stack.

The function invocation process works in the following way:

- The function invocation uses a specific name; only functions registered under that exact name are considered.
- A set of all *matching functions* is defined: these are all the functions (with the correct invocation name) for which the registered parameter types match the runtime types of the corresponding stack elements at call time. E.g. in the example above, that function `def::foo` is a matching function if

and only if the top stack element has type **`std::string`** or a sub-type thereof, and the stack element immediately below has type **`std::int`** or a sub-type thereof.[4]

- The matching functions are ordered by *precision*. Let f be a function registered with parameter types $r_m, r_{m-1}, ...r_1$ (in stack order, r_1 is top-of-stack), and g be a function registered with parameter types $s_n, s_{n-1}, ...s_1$. f will be said to be *more precise* than g if and only if all of the following hold:
  - $m \geq n$ (i.e. f is registered with at least as many parameter types as g)
  - For all $1 \leq i \leq n$, type r_i is a sub-type of s_i.
- If one of the matching functions is more precise than all other matching functions, then that function is called. Otherwise, an error occurs.

The following important points must be noticed:

- Precision order is partial. Two given functions are not necessarily comparable, i.e. neither being "more precise" than the other. The invocation process does not require that all matching functions be comparable to each other, but that one can be compared to all others, and found to be more precise than all others.
- A failure will be reported if there is no matching function, but also if there are several and none is more precise than all the others.
- Since sub-typing is acyclic (except that every type is deemed to be a sub-type of itself), the only way for two functions f and g to be such that f is more precise than g and g is more precise than f at the same time, is to have f and g registered with the exact same parameter types. This situation is explicitly forbidden: any attempt at registering a function with the same name and parameter types as an already registered function triggers an error.
- If a function is registered with n parameter types, and the stack contains fewer than n elements at call time, then that function is not a matching function.
- The parameter types used for registration do not necessarily exhaust all the actual function parameters. A function registered with two parameter types may use more than two parameters. Moreover, registration says nothing about how the number and types of values a function may return (i.e. leave on the stack when exiting).

This process works best when registered functions use the same patterns. For instance, it is expected that most functions in a given application will work like ordinary methods as in classic OOP, i.e. be dispatched based on the type of a single parameter, which will be "the object on which the method is invoked". To allow functions to be used without undue collisions, even if the same names are used, it is best if all such method-like functions are registered such that the owner object is the top-of-stack (i.e. rightmost parameter in the list).

[4] In that specific case, **`std::int`** cannot have sub-types, but **`std::string`** can.

5.3 Immediate Functions

An *immediate function* is a function which is registered with no parameter types, and a special "immediate" flag. The role of immediate functions is to be invoked as soon as they are encountered in the source code, during interpretation; this is how additional syntax is defined.

6 Interpretation Syntax

Interpretation is the process during which source code is translated into instructions to execute. The source code may itself trigger the immediate execution of the functions which it just defined; they then run in the context of the interpreter. *Compilation* is a separate step that may optionally occur when triggered by the interpreted code, or implicitly at the end of interpretation, or not at all; this is covered in a later section.

The source code syntax is defined in terms of interpreter behaviour. Like in Forth, there is no formal syntax that is parsed into a tree; instead, the main interpreter is a simple loop, which is then extended by *immediate functions*, which are functions that are executed immediately when their name is encountered, and interact with the source stream to implement all extra syntax. User code can define its own immediate functions, and thereby process the source code in arbitrarily extensible ways.

6.1 Function Building

At any point, the interpreter is *building a function*, i.e. accumulating instructions into an as-yet incomplete function. Building contexts nest: at any time, a new building context may be opened; the previous one will be restored when the new context is closed. Closing the context yields the function.

Some building contexts are said to be *automatic*: whenever some instructions have been accumulated in an automatic context and there is no outstanding flow control structure, the context is closed (which creates the corresponding function), cleared, and reopened; the function which was just created is then executed.

For a non-automatic building context, the new function is registered as soon as it is created. For an automatic building context, the new function is not registered and does not have a name; it is executed right away, then discarded. Note that the automatic context is cleared and reopened *before* running the newly created function: this allows that function to populate the building context with new function elements.

In Forth, the interactive system has two states, “interpretation” and “compilation”. In interpretation mode, typed words are executed immediately, while in compilation mode, they are recorded in the currently-built function (“word”, in Forth terminology). The interpretation/compilation duality complicates the description of the language, in that many words have different semantics depending on the current state. In plain Forth (not counting some non-standard extensions), states do not stack, and you cannot define a sub-function within a function. Moreover, flow control structures are not available in the interpreter.

In T1, the term “compilation” is reserved for a distinct process, described later on. Thus, “interpretation” is used for all activities related to source code processing. The automatic building contexts are functionally equivalent to the Forth “interpreter”, except that they allow all flow control structures, and do not require special semantics.

Functions are made of the following formal instructions:

- CALL: invoke a function with a specific name (normally a qualified name token).
- CONST: push on the stack a given value (a reference).
- GETLOCAL: push on the stack the value currently held in a specified local variable.
- GETLOCALINDEX: push on the stack the value currently held in a specified local variable (among an array of locals, by index).
- PUTLOCAL: pop a value from the stack and write it into a specified local variable.
- PUTLOCALINDEX: pop a value from the stack and write it into a specified local variable (among an array of locals, by index).
- REFLOCAL: push on the stack a reference to a locally allocated instance.
- REFLOCALINDEX: push on the stack a reference to a locally allocated instance (among an array of locally allocated instances, by index).
- RET: exit the current function, returning control to the caller.
- JUMP: unconditional jump to another point in the sequence of instructions (within the same function).
- JUMPIF: conditional jump to another point in the sequence of instructions: the top-of-stack is popped and must be a boolean value; the jump is taken if that value is **`true`**.
- JUMPIFNOT: conditional jump to another point in the sequence of instructions: the top-of-stack is popped and must be a boolean value; the jump is taken if that value is **`false`**.

Local elements are statically indexed, i.e. which local variable or instance, within the current activation context, is used in a PUTLOCAL, GETLOCAL or REFLOCAL, is a question which is decided at the time the instruction is added to the currently built function. For GETLOCALINDEX, PUTLOCALINDEX and REFLOCALINDEX, the location and length of the sequence of local variables or instances are also decided at function building time; the index is popped from the stack at runtime, and compared with these bounds to prevent out-of-bounds accesses.

The JUMP, JUMPIF and JUMPIFNOT opcodes are together called the "jump opcodes".

6.2 The Interpreter Loop

The interpreter loop is described in pseudo-code as follows:

1. Read the next token from the source code stream. If there is no next token (end of source stream), exit (the interpretation process is finished).
2. If the token is a numerical constant or a literal string, then add a CONST opcode to the current function for that value (for character strings, this implies creating the instance that contains that string, and using the reference to that new instance as value); then jump to step 6.
3. The token is a name. If the name is unqualified:
   (a) If the name matches that of an accessor for a local variable or locally allocated instance, then the corresponding opcode (GETLOCAL, PUTLOCAL...) is added to the current function; then jump to step 6.
   (b) If there is a currently defined alias for that name, then convert the name into the qualified name to which the alias points.
   (c) Otherwise, convert the name to a qualified name by adjoining the current namespace.
4. The name is qualified. If there is a currently registered immediate function under that name, invoke it immediately, then jump to step 6.
5. Add a CALL opcode to the current function, for the qualified name.
6. While all of the following hold:
   - the current building context is automatic;
   - the current building context is not empty;
   - the current context does not have outsanding flow control structures;

   finalize the current context into a function f, reinitialize the context into a new empty function builder, and execute the function f.
7. Jump to step 1.

For this description, in step 1, we assume that the source code is a single input stream. In practice, the interpreter will handle several successive source files, one at a time, with a new interpreter loop for each file. Building contexts are not conserved across files, so any function whose building has started must be finished by the end of the same file.

Numerical constants are the two boolean values (**`true`** and **`false`**), character constants, and number constants. Character constants are words that start with a backquote character ("`"), while number constants start with an ASCII digit, or a plus or minus sign followed by an ASCII digit. Literal strings start with a double-quote character ("""). The syntax for numerical constants and literal strings was described in section 2.

TODO: The constant parsing process will be made pluggable, so that new arbitrary constant formats may be defined, normally distinguished by suffix. For instance, when "big integers" are implemented, they will use numerical constants with a "`z`" suffix. Similarly, floating-point constants will use a dot symbol ("`.`") and the usual exponent notation. In all generality, there will be a sequence of registered functions that are invoked in due order until one returns that it could understand the format; the last one will apply the rules for integer types.

In step 3, a PUTLOCAL into local variable `x` is obtained with the name "`->x`", without a space between "`->`" and "`x`". If a space separates both parts, then a PUTLOCAL will also be obtained through a much different road, the name "`std::->`" being itself an immediate function that implements an extended syntax for writing into local variables.

In step 4, the "immediate" flag is an extra information attached to the function when registered. A normal CALL opcode that targets an immediate function name (i.e. adding a call to the immediate function in the currently built function, rather than calling the immediate function immediately) can be obtained with the "quoting function" (`'`, described later).

In step 6, a loop is used because execution of the current function may again add opcodes to the current automatic context. Note that the context is cleared and reinitialized after finalizing the current function, but before calling it, precisely so that new opcodes may be added to the context without being discarded. The call to the built function is direct and does not use the name and lookup process (the function is not actually registered).

The meaning of "outstanding flow control" will be explained in section 6.5.

6.3 Nested Interpreter

While a given interpreter loop can work over nested building contexts, a common pattern in the T1 syntactic constructions is the use of a nested interpreter loop. This is normally triggered by the opening parenthesis token ("`(`"):

- A new builder context is created and opened. This is an automatic context, i.e. with immediate execution of instructions.
- A new, empty data stack is created.
- The nested loop runs until it reaches a closing parenthesis token ("`)`"), using the new data stack. If the end of the source stream is reached before obtaining that closing parenthesis, an error is raised.
- When the nested loop exits, the caller obtains the contents of the data stack which was created for the nested loop. If, at that point, the current builder context is not the automatic context that was created for the nested interpreter, or that context is not empty (because of an outstanding flow

control structure), then an error is raised. Otherwise, that context is removed, and the previous context, which was active when the opening parenthesis was encountered, is restored.

One case is when the interpreter loop encounters the opening paranthesis in its normal processing loop, at step 3 (specifically, when after applying aliases, the qualified name is "`std::(`"). In that case, a nested interpreter loop is launched, in the conditions described above. When that loop exits, the contents of its dedicated data stack are used for that many CONST opcodes added to the current builder context.

Other uses of a similar construction are for function and type declarations.

6.4 Function Declaration

To declare (and define) a new function, the "`:`" function is used. Here is an example:

```
: fact <export> (u64)
    # ...
```

The behaviour of "`:`" is as follows:

1. Get the next token from the source stream; if it is a literal string, then the string contents are the name under which the function shall be registered; otherwise, it shall be a name token. In the latter case, if the name is unqualified, then it is converted to a qualified name:
    - If there is a defined alias for the name, then the qualified name to which the alias points is used.
    - Otherwise, a qualified name is made by adjoining the current namespace to the parsed raw name.
2. After the name may follow one or several of the following qualifiers:
    - "`<export>`": the new function will be added to the export list of the current namespace (see the section 6.8 for details).
    - "`<immediate>`": the new function will be registered as immediate when finished building.

    Order of appearance is not significant for qualifiers; if the same qualifier is applied several times, this has the same effect as a single instance of the qualifier. Unknown qualifiers trigger an error.
3. An opening parenthesis ("`(`") terminates the list of qualifiers, and starts a new, nested, automatic building context, as described in section 6.3. The stack contents upon exit of the nested interpreter loop are then used as the list of parameter types for registration of the new function. If any of the values is not a type (an instance of "`std::type`"), then an error is raised.

    If the new function is immediate, then the list of types shall be empty (immediate functions are registered with an empty parameter list); otherwise, an error is raised.

4. A new building context is created for the new function. The function name (qualified), flags (exported, immediate...), and parameter types are stored in that context, and will be used when the context is closed.

Since a new building context was created, subsequent actions of the interpreter loop will add opcodes to that builder, hence contributing to the code of that new function. The new function is not registered until its building context is finalized. This is normally triggered by the immediate function "`;`".

The "`:`" function is immediate, thus allowing the declaration of a new function while another function is being built (this contrasts with Forth, where nested function declarations are not supported, and "`:`" is not immediate, since it is supposed to be invoked only from the interpreter). These nested functions do not have any scoping hierarchy or similar features: a nested function has the same visibility as any other, and it cannot access the local variables and instances of the outer function. This feature is mostly a syntactic convenience.

6.5 Flow Control

The jump opcodes are added with dedicated immediate functions, that use a *control-flow stack* which is managed by the builder context. That stack is separate from the data stack. It contains "origins" and "destinations":

- An *origin* represents a jump opcode that has been added to the current builder, but still needs to be resolved to its destination.
- A *destination* represents an opcode of any type that may become the target of a jump opcode.

The control-flow stack is a powerful concept imported from Forth. In Forth, it is implementation-dependent whether the control-flow stack uses the data stack, or is separate; in T1, the control-flow is separate and bound to the builder context, which avoids any issue with nested function builders.

The concept behind the control-flow stack is that a jump opcode is first added, then resolved; resolution occurs either when the target destination becomes known, for a *forward jump* (the target is beyond the jump, and thus added later on), or when the jump opcode itself is added, for a *backward jump* (the target is before the jump, and already present at the time the jump opcode is added):

- An origin is pushed when adding a forward jump.
- An origin is consumed when adding the target for a forward jump.
- A destination is pushed when adding the target for a backward jump.

- A destination is consumed when adding a backward jump.

At any time, the builder has a *current address*, which designates the next opcode that will be added. Thus, whenever an origin or destination is pushed, it designates the opcode that will be added next.

Consider for instance the classic "if" construction. As per the Forth tradition, it syntactically looks as follows:

```
... # some code that pushes a boolean value
if
    ... # executed if the boolean is true
else
    ... # executed if the boolean is false
then
```

This code has two forward jumps:

- a forward JUMPIFNOT at the position of the "`if`", to consume the boolean value and skip the first code chunk if the boolean falue is `false`; that jump targets the second code chunk, just after the "`else`";
- a forward JUMP opcode at the position of the "`else`", so that after execution of the first code chunk (when the boolean was `true` and the JUMPIFNOT was not taken), execution skips to the code that follows the final "`then`".

The behaviour of the three immediate functions is as follows:

- "`if`": push the current address as an origin, and add a JUMPIFNOT opcode (thus, the "origin" qualifies that newly added opcode).
- "`else`": push the current address as an origin, add a JUMP opcode, swap the two top elements of the control-flow stack, and pop the top element (it should be an origin) to resolve it against the current address.
- "`then`": pop the top control-flow stack element (it should be an origin) to resolve it against the current address.

Thus, the "`if`" adds an as-yet-unresolved forward jump, which is pushed as an origin on the stack; "`else`" adds another forward jump, also pushed as an origin on the stack, and resolves the first jump to the opcode that will immediately follow the second forward jump; "`then`" resolves the second jump opcode.

Since origins and destinations are organized as a stack, this naturally supports nesting flow structures.

A builder is said to have *outstanding flow control structures* when its control-flow stack is not empty. In such a case, an automatic builder does not finalizes itself.

The use of the control-flow stack to decide whether an automatic builder context finalizes and executes the current function or not, allows the use of flow control structures in code meant for immediate execution. This contrasts with Forth, where (normally) you cannot use flow control in "interpreter mode".

When a function builder is finalized, its control-flow stack must be empty, otherwise an error is raised. Also, all jump opcodes must have been resolved. Normally, resolution is done by consuming items on the control-flow stack; however, since items on the control-flow stack can be explicitly duplicated and dropped, the two conditions "stack is empty" and "all jumps are resolved" are not necessarily synonymous.

All functions end with an implicit RET opcode. Thus, if the current address was used to resolve a forward jump, but no opcode was added afterwards, the jump targets that implicit RET.

6.6 Type Declarations

A structure type is defined with the `std::struct` immediate function. This function parses the new structure name, and then the structure fields and embedded sub-structures. Here is an example of such a declaration:

```
struct foo <export>
    x   int              # reference field of type std::int
    vx  12 u8            # embedded array of 12 std::u8 values
    p   bar              # reference field of type def::bar
    q   && bar           # embedded structure of type def::bar
        && qux           # def::foo extends def::qux
    a   (i32 array)      # reference field of type (std::i32 std::array)
    b   && 5 (int list)  # embedded array of 5 instances of (std::int std::list)
end
```

The line breaks and indentation are not significant, and have been set for clarity of the source code only. The comments (starting with "`#`") are similarly not significant.

The example above illustrates the characteristics of the type declaration syntax:

- Elements are declared with the element name, followed by its type.
- If an integer constant lies between the element name and the type, then the element is an embedded array.
- The special name "`&&`" is used to embed sub-structures. It can be combined with an integer for an embedded array of embedded sub-structures.

- If the element name is missing, and the "`&&`" special name appears where a name was expected, then this is a type extension, which combines embedding and sub-typing. The name of the embedded class is also used as the element name.
- When an element type is expected, an opening parenthesis can be used, to create a nested interpreter loop that evaluates to the `std::type` instance to use.

The behaviour of "`std::struct`" is as follows:

1. Parse the next token from the stream. If it is a literal string, then the string contents are the type name; otherwise, the next token shall be a name. In the latter case, if the name is not qualified, then it is converted to a qualified name by applying the currently defined aliases, or adding the current namespace if none of the current aliases applies.
2. Get the type that currently has the specified name. If there is no such type, a new structure type is declared and used. If the type exists but is closed, then an error is raised; otherwise, the existing type will be used.
3. If the next token from the stream is the name "`<export>`", then the type function (the function that returns the "`std::type`" instance corresponding to the new type) will be marked as exported (see section 6.8 for details); otherwise, the next token is pushed back onto the stream, to be read again at the next step.
4. Parse the next token *t* from the stream. If that token *t* is the name "`end`", then the type declaration stops, and the immediate function "`std::struct`" returns. Note that the type is *not* closed.
5. If the token *t* is the name "`&&`", then this is an extension:
   (a) A type reference is parsed. This must be a single type instance T, with no integer count. Type reference parsing is described later.
   (b) A new element is added to the current structure, using the type T, with the name of T as element name. Then go to step 4.
6. The token *t* must be a name. If that name is unqualified, then it is converted to a qualified name *n* by using the current aliases (if applicable), or the current namespace. Otherwise, *n* is set to be equal to *t*.
7. If the next token is the name "`&&`", then that token is parsed (i.e. discarded from the input stream), and the new element will be an embedded sub-structure, or an embedded array of embedded sub-structures; otherwise, the next token is left on the input stream for the next step, and the element will be a field or an embedded array of fields.
8. A type reference is parsed. This may be either a type instance T, or a pair consisting of an integer value *x* followed by a type instance T. In the latter case, an embedded array of *x* elements is defined;

the value *x* must be greater than zero (otherwise, an error is raised). The type *T* applies to the new element (as type of the field, or the embedded structure, or the embedded array element values or embedded structures, depending on the presence of the integer *x* and the initial "**`&&`**" token).

9. Go to step 4.

Parsing a type reference is a sub-process that behaves as follows:

1. Start with an empty list of values. "Adding to the list" means appending a new value at the end of the list.

2. Get the next token *t*. If that token is a numerical constant, then it shall be of type "**`std::int`**" (otherwise, an error is raised); that value is added to the list, then the process loops to step 2.

3. If *t* is an opening parenthesis ("**`(`**"), then a nested interpreter loop is executed, as specified in section 6.3; the output contents of the data stack of that loop are then examined:

   - If the nested stack contains only **`std::int`** values, then these values are added to the list in stack order (top-of-stack is added last); then loop to step 2.
   - If the nested stack contains zero, one or more **`std::int`** values, followed by a single **`std::type`** instance (as the top-of-stack), then these values are added to the list in stack order; then jump to step 6.
   - Otherwise, the stack contents are not valid, and an error is raised.

4. If *t* is a literal string, then the string contents are used as type name *n*. Otherwise, *t* shall be a name; that name is used for *n* (converted to a qualified name with the current aliases and namespace, if necessary).

5. The type of name *n* is added to the list. If that type does not exist, then a new empty, open structure of name *n* is created, and its **`std::type`** instance is used.

6. The list contents are returned.

By construction, the parsing of a type reference can only return a **`std::type`** instance, preceded by zero, one or more **`std::int`** values.

TODO: Allow multi-dimensional arrays. The type parsing mechanism can return more than one integer value. It is unclear whether multidimensional arrays are really a good idea, though: they are merely a syntactic shortcut for computing the index as a multiplication and an addition, since all dimensions are fixed (no "jagged arrays"). Defining multi-dimensional arrays would require making special accessor names, e.g. "**`v@@`**" to make it syntactically explicit that two index values are expected.

The type parsing mechanism allows the use of generics. Consider the two following element declarations, which have similar effects:

```
    x  "(std::u8 std::list)"
    y  (u8 list)
```

In the first case (element **x**), the explicit name of the "growable array of bytes" type is used, while in the second case (element **y**), a nested interpreter loop is used; that loop will first call the **std::u8** function (which pushes the **std::type** instance of unsigned integers modulo 2^8), then call the **std::list** function, which will use the **std::type** instance on the stack as parameter for creating the **std::type** instance for the growable array of bytes.

The second syntax is easier to use, because the nested interpreter loop mechanics will include the automatic qualification ("**u8**" is converted to "**std::u8**" as per the aliases imported from namespace **std**) and be lenient about whitespace, whereas the use of the literal string for **x** requires using the exact type name.

Moreover, using the nested interpreter loop is also more robust: the call to **std::list** creates the type on demand, and, in particular, also creates and registers all functions that operate on growable vectors of bytes. The syntax with a literal string does not perform this task, and thus relies on other constructions in the source code to ensure that the said functions exist.

6.7 Local Variables And Instances

Local variables are slots that can receive a value (i.e. a reference) and that exist within the activation context of a function; they disappear when the function exits. Similarly, *local instances* are object instances that are allocated when a function activation context is created, and meant to be released when the function exits.

The generic syntax for creating local variables and instances mimics that of the declaration of types. It starts with the **local** immediate function:

```
local
    x   int              # reference field of type std::int
    vx  12 u8            # embedded array of 12 std::u8 values
    p   bar              # reference field of type def::bar
    q   && bar           # embedded structure of type def::bar
    a   (i32 array)      # reference field of type (std::i32 std::array)
    b   && 5 (int list)  # embedded array of 5 instances of (std::int std::list)
end
```

For each named element, *accessor names* are created. These names are recognized by the interpreter loop (see section 6.2) when building the function, and converted to the appropriate opcodes. Such names do not exist beyond function building, are not bound to any namespace, and cannot be aliases (they are matched before applying aliases and namespaces). These names are the following:

- For a field of name `x`:
  - `x` returns the current contents of the field.
  - `->x` writes a value into the field.
- For an embedded structure of name `x`:
  - `x&` returns a reference to the structure.
- For an embedded array of references of name `x`:
  - `x@` reads a reference value, using an index (of type `std::int`).
  - `->x@` writes a reference value, using an index (of type `std::int`).
  - `x*` initializes an array instance (of the right type) to provide an array view of the array.
- For an embedded array of embedded structures of name `x`:
  - `x@&` returns a reference to one of the embedded structures, using an index (of type `std::int`).
  - `x*` initializes an array instance (of the right type) to provide an array view of the array.

Several `local` declarations may exist within a function, provided that no local name is reused within that function.

There is no smaller scope than a function. When a local variable or instance is declared, its name becomes usable until the end of the function building, but the corresponding variable or instance is created when the function activation context is created, i.e. upon function entry.

Local variables and instances are accessible only within the function in which they were declared. In particular, if a new function builder is opened without closing the current builder, the syntactically nested function builder is separated from the outer function; it does not have any access to the outer function's local variables and instances, and it may create its own local variables and instances without any restriction on types and names. The "nesting" feature does not have any significance for the built functions.

Since accessor names are resolved syntactically, the types associated with local variables are not used for any registration mechanism. They are still used as filters for write accesses: if a field is declared with type `std::foo`, then only references to such a type, or a sub-type thereof, may be written into the field. This is meant as a way to document intended types for local variable contents. Compliance with such type filters is verified dynamically by the interpreter (upon each actual write access), and statically by the compiler.

When a function is entered, local variables, and local instance fields, are filled with their default values: booleans are `false`, small modular integers are zero, and all other types (including `std::int` fields) are uninitialized. Reading an uninitialized field triggers an exception. Contrary to structures, no accessor names are provided to test a local variable for initialization, or to reset it to uninitialized state: the intent of local variables is to never be read while still uninitialized, and the compiler will refuse to compile functions for which it cannot prove that local variables are never read before being written. This is meant to allow for more optimized usage of local variables, without tests for uninitialized state, at least in compiled functions. This also mimics the behaviour of both Java and C# compilers.

Since references to local instances can be obtained, it is possible to access such instances after the activation context in which they were created has been destroyed. This is permitted in the interpreter (which implies that such local instances may actually be heap-allocated). However, the compiler enforces escape analysis to make sure that such survival does not happen, allowing local instances to be truly allocated within the activation context.

An alternate syntax for declaring local variables uses the immediate function `std::{` (note that the opening brace "`{`" is a special character for the lexer, and thus a name by itself). The folowing:

```
{a b c}
```

declares three local variables of names "`a`", "`b`" and "`c`", respectively. They have type "`std::object`", i.e. can accept any value (reference), and are initially uninitialized.

The special immediate function "`std::->`" can be used to write to several local variables at once, or even to combine declaration and initialization. When that function is executed (i.e. when encountered in source code, since it is immediate):

- If the next token is "`{`", then this opens a list of names, ending with the closing token "`}`". This both declares and initializes local variables.
- Otherwise, if the next token is "`[`", then this opens a list of names, ending with the closing token "`]`". This writes to several local variables, but does not declare them; the local variables must already exist.
- Otherwise, the next token must be the name of an already declared local variable, and this is a write to that variable.

When writing to several local variables at once, they are listed in stack order (rightmost is top-of-stack). The three following constructions thus have identical effect:

```
# Declare and initialize three variables.
->{a b c}
```

```
# Declare three variables, then write to all of them at once.
{a b c} ->[a b c]

# Declare three variables, then write to them one at a time.
{a b c} ->c ->b ->a
```

In the third one, note that "`->a`" is interpreted as the accessor word that writes to the variable "`a`", while "`-> a`" would be parsed as the "`->`" immediate function, that then parses the token "`a`", and adds to the current function the effect of writing to "`a`" (a PUTLOCAL opcode), i.e. the same final outcome.

6.8 Namespaces and Imports

At any point when processing source code, there is a *current namespace* which is used to qualify raw names for which no active alias was found. The default current namespace is "`def`".

The *current aliases* are a mapping from raw names to qualified names. Such mappings are built one at a time, and with *import lists*. An import list is made of all names defined in a given namespace and declared "exported".

The "`std::namespace`" immediate function changes the current namespace:

```
# Switch the current namespace to "foo"
namespace foo
```

The `namespace` function parses the next token, which must be an unqualified name.

When the new current namespace is changed, all the currently defined aliases are cleared, and the import list for namespace `std` is loaded.

Aliases are defined with the "`std::alias`" immediate function. This function parses the next token:

- If the next token is a qualified name `n::r`, then the alias is for the raw name `r` to the qualified name `n::r`.
- Otherwise, the next token must be a raw name `r`. The token that follows must then be a qualified name `n::s`, and the mapping will be from `r` to `n::s`.

Import lists are obtained with the "`std::import`" immediate function. This function parses the next token, which must be an unqualified name. That name is taken to be that of a namespace, and the contents of the current list of exported names from that namespace are added to the current list of aliases. Take care that the list of imported names is the one at the time the **`import`** clause is processed; names later added to the import list of that namespace are not automatically imported.

Name collisions are handled with the following rules:

- A defined alias consists in the following:
  - A *source name*: this is the name *for which* the alias is defined. It is always a raw name.
  - A *destination name*: this is the name *to which* the alias is set. This name is an arbitrary string, but is usually a qualified name. It may also be the special *invalid-name* value, which is distinct from all strings.
  - A *provenance flag*: it is meant to be set for names that have been set explicitly with **`std::alias`**, and cleared otherwise.
- When an **`std::alias`** clause is used to define an alias for raw name `r`:
  - If there is no currently defined alias for `r`, then the alias is defined as specified by the clause; its provenance flag is set.
  - Otherwise, if there is a currently defined alias for `r`, whose provenance flag is cleared, then the alias's destination is set to the destination name provided by the **`std::alias`** clause, and its provenance flag is set.
  - Otherwise, if the currently defined alias for `r`, with its provenance flag set, has the same destination name as the one specified by the **`std::alias`** clause, then nothing happens.
  - Otherwise, the new alias points to a name distinct from the destination of the old alias, and the old alias has its provenance flag set: in that situation, an error is raised.
- When an **`std::import`** clause is used to load an import list, and the import list contains an alias for a raw name `r`:
  - If there is no currently defined alias for `r`, then the alias is defined as specified by the clause; its provenance flag is cleared.
  - Otherwise, if there is a currently defined alias for `r` that points to the same name as the name defined in the import list, then nothing happens.
  - Otherwise, if the currently defined alias for `r` has its provenance flag set, then nothing happens.
  - Otherwise, the new alias points to a name distinct from the destination of the old alias, and the old alias has its provenance flag cleared: in that situation, the alias's destination is set to *invalid-name*.
- Whenever an alias is *used* for raw name `r` (i.e. the raw name `r` was encountered in a syntactic construction, and it is to be transformed thanks to the current aliases), and there is a currently defined alias for `r` whose destination is *invalid-name*, then an error is raised.

- The provenance flag has no influence on alias usage.

Import lists are roughly similar to Java's whole-package imports, e.g. "`import java.util.*;`". They have the same convenience of getting easy access to many names with a single programming clause, but they also share the same compatibility risks: if an import list is later modified by the source package to include more exported names, these new names may enter in conflict with other names defined by the source code or imported from other lists. The mechanism with *invalid-name* and the provenance flag is meant to solve such issues along the following principles:

- The developer is supposed to know what happens in her own namespace. Thus, a conflict between two explicitly defined aliases is a programming error, hence sanctioned immediately.
- Similarly, a collision between an imported alias and an explicitly defined alias is resolved in favour of the latter: an explicit alias has precedence over an imported alias.
- A collision between two imported aliases is not resolved, but does not trigger an immediate exception: the import lists are considered to be out of reach of the developer, and thus may incur collisions that she cannot prevent. However, use of the name on which the collision occurs becomes ambiguous, and thus triggers an exception.
- Redefining an alias identically is always permitted.
- Order of declaration should not matter.

Ambiguous names (from collisions between import lists) can be resolved with an explicit `std::alias` clause, that will take precedence over both import lists.

It is a matter of programming style whether to use import lists or explicit aliases. The import list from `std` is always loaded because it would be very inconvenient to write code without (in that respect, it is similar to OCaml's "Pervasives" module). Functions from other namespaces may be used with explicit namespace names, or explicit aliases, or import lists, or any combination thereof.

Existing syntax favours the declaration of "simple aliases" that map a raw name to a qualified version of the same name; this is what the `<export>` keyword does when defining a function. Nevertheless, other (to be defined) API may be used to make aliases that map raw names to arbitrary strings, both explicitly and through import lists.

6.9 Errors

In all of the previous text, the expression "raising an error" was used many times. An error terminates execution immediately and is not recoverable. It may include an error code for reporting purposes.

There are several models for error handling, notably the following:

- Individual functions may report errors as special values, as in C: for instance, a `read()` call on a file descriptor (on Unix-like systems) returns either the number of bytes that have been read, or the special value `-1`.
- To avoid the need to put error codes in the same space as values, the result may be wrapped into a container that retains whether a value or an error was obtained, and additional syntactic constructions are provided to test for errors and obtain the result; this is how things are done in Rust.
- In languages where functions can return several values, functions may return the result *and* an error code as separate values; this requires a special "no error" error code, that the caller can easily test, as well as a default value to return along with the error code in case of error. Go uses this mechanism.
- Errors may be reported through thrown exceptions, as in Java or C#. Activation contexts are unwinded until a catch mechanism is reached.

All of these mechanisms are imperfect, in particular on small, constrained systems. Error values require extra code to receive them, test for them, and, more often that not, propagate error codes up the call chain. Exceptions tend to allow for a more compact and efficient implementation, in that they keep error handling out of the main processing; however, catching exceptions implies more complicated semantics that make code generation harder, and can increase code footprint.

In T1, a cruder but simpler mechanism is used: any error terminates the whole program, or, more accurately, the whole *module*. One of the points of T1 is to allow compact, efficient coroutines; thus, an application that uses T1 is supposed to be split into several modules that act as coroutines to each other. Each module has its own stack and heap, and modules communicate with each other only through serialized messages. For instance, in an SSL/TLS library, a module written in T1 could handle X.509 certificate validation; it receives the encoded certificate chain, and returns the validation result (notably the public key from the certificate). Any validation failure then cancels the complete module, but not the application. In effect, T1 error management is about concentrating handling at module boundaries. This also maps to a clustering structure in which T1 modules might run on a distributed system.

7 Compilation

Compilation is a step which optionally occurs at the end of interpretation, when T1 is invoked "as a compiler"; it can also be triggered explicitly by the source code itself. Compilation takes as input a list of *entry points* (specific functions), and produces an executable form of these functions and their transitive dependencies. This process is meant to fulfill the following:

- Compiled code is small and self-reliant. It can be invoked and run without requiring access to a bulky runtime system.
- Interpretation features, such as defining types or new functions, are not available in compiled code. Notably, compiled code cannot access type or function names.
- Compiled output should be amenable to integration within applications written in other languages, in particular C.
- The compiler offers strong guarantees on the usage of memory resources by compiled code: maximum data stack depth and maximum storage area for activation contexts (including local variables and locally allocated instances) are computed; and dynamic memory allocation, if supported at all, can be made to occur only in a specific, limited area provided by the caller that invokes the compiled code.
- Compiled code is proven not to trigger any error related to function invocation: whenever a function is invoked, there is exactly one matching function that is more precise than all other matching functions; and accessor functions called on instances that extend the structure on which the accessor was defined find an unambiguous instance on which the access is to be performed.
- Similarly, compiled code is proven never to read an uninitialized local variable, to let a reference to a locally allocated instance escape its activation context, or to attempt to write into a statically allocated instance.

Compilation can work only on a subset of valid codes; notable among the restrictions is that compiled code cannot be generally recursive, since such recursion would prevent computing strong bounds on stack depth.

Banning recursion is controversial, especially since most functional languages instead strive to use recursion to express most of flow control. The two main reasons to forbid recursion in T1 are the following:

- Not allowing recursion means that the call tree is finished, which permits the general flow analysis (described below) to terminate.
- Recursion allocates memory in spaces which are scarce on memory resources. T1 aims at being useful for small embedded systems that have only a few kilobytes of RAM in total; however,

even on bigger systems, stacks are small. For instance, a typical modern desktop system or laptop will have gigabytes of RAM, but the stack allocated for a thread is smaller (8 megabytes by default on Linux). Common sense dictates that if unbounded memory allocation occurs, it should not be done in an area which is a thousand times smaller than the heap, and for which the only detection mechanism for allocation failure is `SIGSEGV`.

In a future version, *tail calls* may be implemented, and tail recursion allowed. In a tail call, the activation context of the caller is released first, and when the callee returns, control is passed back not to the caller, but the caller's caller. If a tail call does not imply undue stack growth, then it won't prevent computing finite bounds on stack depth, and it *should* be manageable by flow analysis.

7.1 Flow Analysis And Types

Flow analysis is the central step of compilation. Consider the following code excerpt:

```
: triple (object)
    dup dup + + ;
: main ()
    4i32 triple println
    "foo" triple println ;
```

The flow analysis starts with the entry point (`main`) and an empty stack. Then, after the `4i32` token, the stack should contain exactly one element of type `i32`. At that point, the `triple` function is invoked. There is exactly one matching function of that name for a stack with one element of type `i32`, and therefore the call is unambiguous.

Analysis proceeds with the `triple` function. Crucially, analysis of `main` is not finished; it will be continued when this call to `triple` is done. Within `triple`, the calls to `dup` and `+` are followed; notably, when the first `+` call is reached, the stack is determined to contain three elements of type `i32`. When the end of `triple` is reached, the stack is back to containing one element of type `i32`. At that point, flow analysis jumps back to the caller (`main`) and can proceed to the next call (`println`) since it is now known that this call is potentially reachable (the `triple` function may return) and also which stack contents to expect at that time. The call to `println` is resolved to the function of that name that expects an element of type `i32`.

Later on, the analysis reaches the second call to `triple` in the `main` function. For that one, the stack contains one element of type `string`[5]. There is still one matching function of name `triple`, and this is the same one as previously (indeed, there's only one `triple` function defined in this example, so only that one may be called). However, the flow analysis of `triple` will be done *again*: everything is done as if that call was a new one.

[5]Strictly speaking, `string` is merely an alias on `(std::u8 std::array)`, but we will use the name `string` for the clarity of the exposition.

In that new call to `triple`, the stack initially contains one `string`; after the two `dup` calls, it contains three `string` elements; then, the `+` calls will be resolved to the function that "adds" strings (it concatenates them into newly heap-allocated string values). That second analysis of `triple` concludes and returns a single `string`. In `main`, the second `println` call is resolved to the function of that name that expects one element of type `string` (not the same one as the one that expects an `i32`).

The salient points of this process are the following:

- The `triple` function has been *registered* with one parameter type, which is generic (`object` matches all value types). It cannot really be called on values of every type; for instance, it cannot be called on `bool` since there is no defined `+` function that works on `bool` values. But it does not matter that the function could *in abstracto* be invoked on values on which it would not work; what counts is whether such an invalid call is actually made in the program at hand. The flow analysis determines that all calls to `triple` will work, and that is sufficient.
- Similarly, `triple` could have been registered with no parameter at all ("`: triple ()`"). During flow analysis, the compiler would still have known that at the time the function is invoked, there is a value on the stack, and the first `dup` call won't underflow. Types for function registration are used *only* to determine which function is called, not to restrict the actual usage of values on the stack[6].
- The fact that each `triple` call has its own analysis avoids type merging trouble. If both calls were the same node in the call graph, then the flow analysis would be faced with calling `+` on a stack with three elements, each being either a `string` or an `i32`. Such a call would not succeed because there is no `+` function that can work over a `string` and an `i32`; the compiler would reject the code as making a call which is potentially unsolvable. In this case, duplicating the `triple` node allows the flow analysis to keep track of the fact that while all three stack elements at this point may be of type `i32` or `string`, they all three have the same type, and cross combinations are not possible.

The node duplication means that, as far as flow analysis is concerned, the "call graph" is a *call tree*.

> Duplication of nodes for function calls is what makes all function "generic" in the Java or C# sense. But since the analysis is done only deductively, i.e. based on what may be on the stack at that point of the program, there is no need for a syntax to express what type combinations are allowed. Again, T1 does not care whether a given function could work on all input values that may exist in the universe, only that it would work with what may actually be present on the stack at the time of the call.

Type merging may still occur because of jump opcodes. For instance, consider this function:

```
: muxprint (object object bool)
    if drop else swap drop then println ;
```

[6]It is still good software engineering to register functions with exactly the parameters that it is going to use, if only for better source code readability.

Suppose that the top three elements for a call to **muxprint** are values of type **A**, **B** and **bool**. In the built function, the call to **println** can be reached from two points: this could follow the "**swap drop**" sequence (the boolean was **false**, the value of type **A** has been dropped, the stack now contains an object of type **B**), or be reached through the jump that is implicit in the **else** construction. In the latter case, the top of the stack will be a value of type **A**.

Thus, flow analysis will consider that when **println** is called, the stack may contain one element which is of type **A** or of type **B**. The call will be accepted only if it is solvable in both cases. If the two cases are solvable, but lead to distinct functions, then both functions will be analyzed, each with its own context.

For the purposes of flow analysis, all individual conditional jump opcodes are considered independent of each other. This means that the following cannot be compiled successfully:

```
: foo (bool)
    ->{x} {y}
    x if 42 ->y then
    x if y println then ;
```

Indeed, this function uses the provided input value (stored in the local variable **x**) to decide whether to put the integer **42** in the variable **y** (first "**if**" clause), and whether to print the contents of the **y** variable (second "**if**" clause). The compiler does not notice that both jumps use the same control value; instead, it considers that the jumps are independant of each other, and, in particular, the first jump may be taken, thus skipping the initialization of **y**, while the second would not, leading to the read of the potentially uninitialized variable **y**.

The idea that conditional jumps are independent of each other has been borrowed from Java. Indeed, with the equivalent Java code:

```
static void foo(boolean x) {
    int y;
    if (x) {
        y = 42;
    }
    if (x) {
        System.out.println(y);
    }
}
```

Java compilation fails with the error "variable **y** might not have been initialized".

This is not considered a great restriction. In practical Java development, that kind of error occurs mostly when adding debug code to an existing function, activated with a global "debug" flag.

The notion of *type* used for the flow analysis is the combination of the **`std::type`** for the value, and the *object allocation point*. An object allocation point is one of the following:

- static allocation (conceptually, in ROM, thus non-modifiable);
- the heap;
- a specific local slot within the activation context of a specific function call, i.e. a node in the call tree.

This information is the basis for the escape analysis (making sure that instances allocated in activation contexts are not reachable after the called function has returned) and for the verification of constant objects (static allocation corresponds to **`const`** definitions in C, and thus normally end up in non-modifiable memory).

Apart from the stack contents, the types of local variables at any point in a given function (for a given activation context, i.e. within a node in the call tree), is also maintained. A special **`nil`** type is used for uninitialized local variables; any attempt at reading **`nil`** is rejected at compilation time.

Types of values written in object fields are tracked. The container types are distinguished by allocation point, but all instances with the same allocation point use the same tracking. Therefore, writing a value of type **`int`** in the field **`x`** of a heap-allocated object of type **`T`** implies that, from the point of view of the flow analyzer, all objects of type **`T`** that are heap-allocated may contain, at all times, a value of type **`int`** in their **`x`** field.

Note that any merging may enrich the list of possible types in a stack slot, local variable or object field, and trigger further flow analysis for all parts of the call tree that depend on it.

7.2 Constraints

The following constraints are enforced by the flow analyzer; any violation implies a compilation failure:

- The call tree must be finished.
- At every merge point, the stack depth is the same for all code paths leading to that point.
- No merge between basic types (booleans and small modular integers), or between a basic type and a non-basic type, may occur, whether on the stack, in local variables, or within fields of a structure type.
- When a local variable is read, it may not contain **`nil`** (which marks the uninitialized state).
- No write to a field of an object with static allocation may happen.

- Whenever a value has an allocation point tied to a given node N_1 in the call tree, and it is written in a field of an other object, then that other object must have an allocation point tied to a node N_2, and N_2 must be either equal to N_1, or a descendant of N_2 in the call tree.
- When a function returns, the stack contents must not contain any value whose allocation point is the node of that function in the call tree.
- For every function call, all possible combinations of types on the stack at that point must be solvable, i.e. lead to a single most precise function call.
- When a field accessor is invoked, the field must be unambiguously located for all possible types of the owner object.

Note that some special functions do not return (e.g. **`std::fail`**). This is detected during flow analysis. As such, some opcodes may be unreachable; these will be trimmed during code generation.

> Each basic type may be merged only with itself because basic types have specific storage requirements, that may differ from those of "normal" values (which are pointers). In the generated code, values of basic types may be passed around on a different, dedicated stack, or different registers. Similarly, an object field declared with type **`std::u8`** should correspond to a one-byte slot in the memory layout; a feature of T1 is that object layouts are predictable, so that they can be accessed from C code.
>
> The finiteness of the call tree is enforced with nested call counters: when a node is entered that corresponds to a given function f, the counter for f is incremented; it is decremented when leaving f. If the counter goes over a given threshold, then compilation stops with an explicit message. This necessarily detects all infinite trees, since there are only a finite number of functions, each with a finite number of opcodes: an infinite tree can be obtained only through infinite recursion.
>
> Some finite recursion is still tolerated. This allows for some cases where the same generic function is used for several levels of a nested structure, but with distinct types that guarantee against unbounded recursion.

7.3 Code Generation

Code generation occurs after flow analysis has completed successfully. How code is generated depends on the target type; the compiler may produce portable threaded code, or native code, or WASM, or anything else. Generated code includes all functions that are part of the call tree; other functions are automatically excluded.

A generic *function merging* process occurs during code generation. In the flow analysis, functions were duplicated: the same piece of code may yield several distinct nodes in the call tree. When generating code, these nodes may be merged back. This is subject to some restrictions and subtleties:

- Some merge operations may not be feasible. In our example with the **`triple`** function, one of the nodes works on **`i32`** values while the other uses **`string`** values. In generated code, these values use different storage techniques (e.g. stack slots of different size, or different registers), which may preclude merging.
- Even when function merging is possible, it may be undesirable for performance: for instance, the unmerged function may have only simple calls (each **`triple`** function node calls a single well-defined **`+`** function), while the merged function may need a type-based dynamic dispatch (if the **`i32`** and **`string`** could be merged, the **`triple`** function would have to look at the runtime type of the values to decide which **`+`** version to call).

In general, in T1, we aim at code compacity, hence apply merging whenever possible. A future annotation will allow to explicitly tag some functions as prohibiting non-trivial merging, i.e. when the relevant types are not all strictly identical. This would reproduce the trade-offs usually seen in C with **`inline`** functions.