A Formal Specification of a Programming Language: Design of Pit

Leif Pedersen <bilbo@hobbiton.org>
Hassan Reza <reza@cs.und.edu>
University of North Dakota
Department of Computer Science
Streibel Hall Room 201
3950 Campus Road Stop 9015
Grand Forks, ND 58202-9015 USA

1 Abstract

Formal specifications and supporting tools are effective at improving the quality and correctness of a software system. A language is usually simple once understood, but communicating this understanding to another person can be difficult, perhaps because a new language often represents a new paradigm. This communication is particularly important when developing a new language; the compiler and other tools are still under development, so learning by doing isn't always possible, and yet to correctly implement the compiler, a solid understanding of the language is necessary. We use Z notation to formally specify part of Pit, which is a general-purpose programming language that we are currently developing. The main idea behind Pit is to create a language where the programmer can choose between allocating memory manually by using statically-typed "primitive" variables or allowing the compiler to insert memory management code automatically by using dynamically-typed "auto" variables. This feature, in turn, allows a programmer to choose between automatically ensuring that there are no buffer overrun or integer overflow vulnerabilities in the code. Information about Pit is available at http://pit.devpit.org/.

2 Introduction

There are a number of motivations for applying a well-established formal method, any of which is considered a viable alternative solution to the informal and ad hoc practice of software design[8]. The application of a formal method to specify a language helps prevent ambiguity, inconsistency, and incompleteness.

A compiler is a large and complex piece of software; it translates a program written in one language such as Pit into a program in another language such as assembly code, machine code, or perhaps C. One of the main characteristics of any compiler is to preserve the semantics of the program being compiled[7]. Therefore, developing correct compilers that can generate faithful target code without introducing any errors is critically important. We present a Z specification[5][6] to formally specify part of Pit[11] to help us design this part of the language before we begin work.

Pit is a general-purpose programming language that we are currently developing. The goal is to create a language where the programmer can choose between allocating memory manually by using statically-typed "primitive" variables or alternatively allowing the compiler to insert memory management code automatically by using dynamically-typed "auto" variables. This allows the programmer to choose between automatically ensuring that there are no buffer overrun or integer overflow vulnerabilities in the code at the cost of efficiency and writing more efficient code at the cost of maintainability and more opportunities for bugs. The innovation is that because the same language supports both, the programmer can choose to write code both ways in the same function, rather than choosing between one or the other for the entire module. This is useful for areas besides security; for example, managing a data structure using automatic memory allocation is also much easier.

The core language and primitive variables are similar to C, and the auto variables are similar in many respects to variables implemented by scripting languages such as Perl or Python. Primitive variables are simply the classical variable types provided by C: integer, float, string, struct, and function, plus trivial variations such as characters, pointers, signed or unsigned integers, etc, with identical semantics. Auto variables are implemented syntactically as another type of variable. What this means is rather than specifying that a variable is an int, float, etc, the "auto" keyword is simply used instead. Semantically, auto represents a meta-type, indicating that the type of the variable is dynamic and determined automatically during run-time by the value assigned to it. Memory for auto variables is allocated and freed automatically. The types that can be stored in an auto variable are integer, float, string, array, hash (instead of struct), and reference (instead of pointer). (In Pit, "strings" are homogeneous sequences and "arrays" are heterogeneous sequences.) Automatic memory management includes the subvalues stored in auto variables, such as elements in arrays and hashes, and referents of references. The compiler implements auto variables by translating every operation on an auto variable into a function call to a supporting library. Pit is self-hosting, in the sense that the compiler tools and supporting libraries for the language could all be written in Pit with no tools written in other languages. This is possible because primitive variables are used to construct the infrastructure required by auto variables and because no virtual machine is used.

While the semantics for primitive variables in Pit are easily described by comparing to C variables, the semantics for auto variables require some explanation. Conversationally, it is convenient to say that they work similarly to variables in Perl or Python, but this is informal and somewhat inaccurate at best. Intuitively, it is a good starting point, but to describe exactly what they are, we wrote a formal description of exactly what an auto variable is. To that end, we use Z notation to describe how they behave and, to some extent, how they are implemented.

3 Related work

The formal specification of a programming language*[3][4]* covers syntax and semantics, which are the main constraints of any language. The construction of a program is described by the syntactical constraints of the language. The syntax is typically defined through a grammar that describes the rules according to which programs must be composed.

Our work is related to the work by Gray et al*[9][10]*. In ***[9]***, they used MetaPRL formal programming environment to write a complete compiler for ML-like language. In ***[10]***, the authors used formal methods to specify programming languages; their main focus is on the reusability of formal specifications in specifying the programming languages.

Programming languages which have been designed with one of the various formal methods have better syntax and semantics, fewer exceptions and are easier to learn. But despite the obvious advantages, attribute grammars, axiomatic semantics, operational semantics, denotational semantics or any of the other most widely used formal methods for programming language description have not gained popularity and general use[9][10]. One reason might be that semantics is much more difficult to describe than syntax and semantic descriptions are not easy to read. On the other hand, a more compelling reason might be that they lack modularity, extensibility and reusability. Though we are not addressing any of these issues in our study, we have created a formal semantic specification for part of Pit.

4 Examples of Pit syntax and translation to assembly

If auto variables could only store simple values, such as integers, floats, and references, they would obviously not be particularly useful. Their power becomes apparent when they are compounded to create a complex data structure. Creating a complex data structure using auto variables is trivial. To do so, simply assign an array or hash to an element of an array or hash, as follows.

```
1.  ...
2.  private auto $root;   // Declare $root as a local variable
3.  $root = array{};      // Assign an empty array to it
4.  $root(0) = hash{};    // Assign an empty hash to the zeroth element
5.  $root(0)("key0") = "array element 0, hash element key0";
6.  $root(0)("key1") = "array element 0, hash element key1";
7.  $root(1) = hash{};
8.  $root(1)("key0") = "array element 1, hash element key0";
9.  $root(1)("key1") = "array element 1, hash element key1";
10. ...
```

Notice how concise this code is because there is no need to explicitly tell the compiler how much memory to allocate for the array or hash tables, when to free the memory, or what the type of each value is. On line 2, we create an array and assign it to $root, which dynamically assumes the array type. The memory for the array is initially allocated with a size of zero, and later it is extended as necessary. On lines 3 and 6, we create hashes and assign them to elements of the array $root. Since the array was previously too small to hold the elements, it is automatically extended to a length of one, and then to a length of two. Similarly, the elements in the hashes need not exist before a value is assigned; they are created automatically as necessary. For comparison, here is similar logic using only primitive variables.

```
...
private array(struct("key0" array(char, 64), "key1" array(char, 64)), 2) $root;
$root(0)("key0") = "array element 0, hash element key0";
$root(0)("key1") = "array element 0, hash element key1";
$root(1)("key0") = "array element 1, hash element key0";
$root(1)("key1") = "array element 1, hash element key1";
...
```

Notice that unless we invest a lot of time and code in manually allocating the memory with a dynamic size, we lose the ability to extend the array dynamically. Also, the structs are not extensible at all, and there are several opportunities for a buffer overrun vulnerability if the input strings come from a distrusted source. These are the same problems that all C code must deal with. However, the striking similarity between the two examples shows two things: first, that choosing between the two approaches is easy for the programmer; second, if the programmer uses the above implementation, an optimizer will be able to eliminate much of the inefficiency by converting patterns found in the first example into patterns shown in the second example when possible.

On the topic of efficiency, the array extension is not as time-consuming as it looks. Behind the scenes, the amount of memory reserved for the array starts at enough for four elements and is doubled every time it runs out of space. This reduces the amount of time spent copying the array to a new location in memory. While this is logically irrelevant to the program's execution, it reduces the amortized time complexity of insertion from $O(n^2)$ to O(n), where n is the number of times one element is added to the array. Additionally, in many cases an optimizer could notice or guess how many elements are needed and pre-allocate the necessary amount of memory. In this case, it would not be hard for an optimizer to transform the entire example into primitive variables by using an array of structs. While beyond the scope of our current work, an optimizer will eventually be an important tool in mitigating the cost of using auto variables instead of primitive variables.

Above, we mentioned that auto variables are implemented by translating every operation into a function call to a supporting language library. More precisely, for each auto variable, the compiler allocates a pointer where the variable's value would otherwise be stored; then each operation on it is translated into a function call so that a run-time library can manage its memory, type, value, ref-count, etc appropriately. For example, if an auto variable is declared in a function, the compiler allocates a pointer in the stack frame and a function call is inserted to allocate a glob of memory on the heap with the initial value of undef (undef is described later). Further operations on the variable result in the compiler inserting more function calls. At the end of the function, another function call is inserted to decrement each auto variable's ref-count or free it if appropriate. This way, it is valid to store a reference to the auto variable outside the function, since the glob need not necessarily be freed immediately when the function returns. Here is the exact translation for part of the example above into intermediate code for a 64-bit machine:

```
// "$root" above is translated into "%base - 64b" here, which was already
// allocated in the stack frame.  After each group of statements, the
// stack is back to its initial state.

// private auto $root;
stack_alloc 64b;                     // Subtract 8 bytes (64 bits) from
                                     // the stack pointer
call $lang.auto.new.undef;           // Constructs a new auto with the
                                     // value undef and ref-count of 1
store <ptr> [ %base - 64b ], <ptr>;  // Pops the pointer to the new auto
                                     // and stores it at %base - 64b

// $root = array{};
stack_alloc 64b;
call $lang.auto.new.array;           // Constructs a new auto with the
                                     // value of an empty array
store <ptr>, <ptr> [ %base - 64b ];  // Pushes the location of $root
call $lang.auto.refcount_inc;        // Inc ref-count because a binary
                                     // operator call decrements the
                                     // ref-count of both inputs
call $lang.auto.binop.assign;        // Perform the assignment, store the
                                     // result (a temp auto holding the
                                     // return value of the assignment)
                                     // in the second parameter's position
```

```
stack_free 64b;                      // Discard pointer to $root
call $lang.auto.refcount_dec;        // Free the temp auto
stack_free 64b;                      // Discard pointer to the temp auto
```

Although the translation looks somewhat obfuscated, it is no more so than the assembly output from a conventional C compiler, and this is easy output for a compiler to generate.

Notice that since a glob may be pointed to in multiple places if references to it are created, and therefore it cannot be reallocated with a larger size (because this may require relocating the glob). This means that the glob’s value element must store a pointer to the value rather than the actual value, since the value must be reallocated if it changes in size. This extra layer of indirection allows, for example, string buffers that are stored in auto variables to be extended automatically as necessary rather than allowing a buffer overrun. It also allows for certain improvements to efficiency that we describe later.

5 Z specification

At this point, we have established some informal familiarity with what auto variables are. Now we will describe formally how auto variables are allocated, how their dynamic type and value are tracked, and how operations on them behave.

5.1 Justification

A formal specification of the semantics of auto variables is useful in several ways. Most obviously, it allows us to reason about what is specified before implementation so we have a clear idea of what to implement before we start. Later, it is also useful as an aid in communicating what we implemented so that other people can understand how to use the language. It also builds confidence in the language as a possible tool for use in a system where safety or reliability are important. Another benefit is it encourages portability of code written in Pit because if someone in the future decides to improve or reimplement the Pit compiler or the accompanying libraries, he can return to this formal specification for a clearer understanding of what must not change. This way existing code written in Pit is less likely to encounter portability problems between the old and new compilers.

5.2 Conventions

Z notation directly represents nothing more than relations between mathematical sets; there is no inherent sequential meaning to the notation. The representation of a sequential system is introduced by certain conventions that represent the state of a system (or subsystem, such as an abstract data type) that allow us to write relations between a set that represents the state of the system before an operation is executed and a set that represents the state of the system after an operation. The common convention for this is to decorate the names of the after-states with a single-quote. Inputs to such an operation are decorated with a question mark, and outputs are decorated with an exclamation point.

The specification that follows uses the notation for abstract data types extensively. An ADT’s state is represented with a schema, and variables in other schemas may be declared with the ADT’s name as the type. Strictly speaking, the name of the ADT represents the set of all possible states for the ADT, and the declared variable represents one state in that set, just as a variable declared as an integer represents one integer in the set of all possible integers. (For more information on representing abstract data types with Z notation, see ***[5][6]***.)

Operations on ADTs are named with the name of the ADT, followed by an underscore and the name of the operation, and to be unambiguous, underscores are only used in schema names that represent operations on ADTs; CamelCase is used otherwise. When referred to in text, a schema's name is always capitalized. Because plain Z is not object oriented, we simply use this naming convention to associate operations with the structures they must operate on. We decided not to use extensions that provide object orientation because we do not need any advanced features of object orientation. Also, the implementation cannot be object oriented as it must ultimately be called from assembly language by code emitted from a compiler; therefore, using an object oriented extension to Z would widen the gap between the specification and implementation with little benefit.

Pit supports exceptions, so we had to introduce a convention for representing exceptions. While the exact Pit syntax for representing exceptions is not discussed here, the accompanying libraries must handle errors through exceptions. To represent this, we simply use an output parameter called "exception!" in all the operational schemas. The caller must check this output parameter after every operation, and if this is set to "null", no exception occurred and execution continues normally; otherwise, the caller must branch to the exception handler. The basic type EXCEPTION represents the possible assignments for this output variable. The actual implementation of exception handling is more elegant, but because it is too low-level to represent here, this is a simple way to represent the logic in the specification.

EXCEPTION ::= null | type_mismatch

5.3 Core structure

GLOBID is a basic type that uniquely identifies each instance of the Glob ADT. This simply represents a memory address. This means that variables declared with the GLOBID type are pointers. We sometimes need this when a variable needs to be shared between several structures in a way that makes it change in all of them if it changes in one of them. This works well because memory addresses really just map integers to data; we represent this here with partial functions mapping GLOBIDs to variables. In summary, a GLOBID should simply be implemented as a pointer.

[GLOBID]

Similarly, we use VALUEIDs to identify each instance of the Value ADT. Each Value has a ref-count that lets us use a copy-on-write implementation (explained later). VALUEIDs should also be implemented as simple pointers.

[VALUEID]

There are two minor things about the specification that are slightly nonstandard. First, some schemas are defined after they are used; we did this because it is significantly clearer to explain the schemas in the order they are presented, and it has no impact on their meaning. Second, unfortunately there is no standard notation for comments within a schema; we use a common convention used in ***[6]***, though, which is right-justified square brackets here above or to the right of what it describes.

The Glob schema is the central part of this specification. It describes a single variable by encapsulating what the current type of the variable is along with its value. Glob is an abstract data type that represents a data structure pointed to by one or more auto variables. It can represent an integer, fraction, string, array, hash, reference, or function-reference. Every auto variable is simply a GLOBID stored on the stack. Every Glob must have at least one reference on the stack, in a data segment, or in another auto variable (when the GLOBID for a Glob is deleted, the Glob is freed).

Undef is a special value with a type distinct from all other types. Normally it means that no value has been assigned to the variable, but undef can also be explicitly assigned, if desired, as a way to represent a null-value. It is equal to itself and no other value, so code can explicitly check for the undef value by comparison. In boolean context, it always evaluates to false. Any other operation on the undef value causes an exception. (When dealing with strings, often C code will use the null-pointer as a distinct value from the empty string. Undef is useful in the same way, however it is also useful in more contexts than just string manipulation.) This is a rather strange data type, because it represents only one possible value; therefore, if a variable's type is undef, its value is undef as well. When implemented, this means that apart from the Glob, no memory needs to be allocated for this value. A null-pointer for the value is sufficient to describe undef. For the purposes of our specification, we define UNDEF to be a basic type with one possible value. The global variable undef is always set to that value; this variable is simply a notational convenience.

```
[ UNDEF ]

│ undef : UNDEF
```

Each Glob has a ref-count which, like in many other languages, lets us automatically figure out when to deallocate the Glob. Here, the ref-count for a Glob tracks the number of occurrences of its GLOBID. Do not confuse these with the ref-counts associated with a Value, which track the number of Globs referencing its VALUEID. (There are two purposes for ref-counts: to track how many auto variables reference a particular GLOBID and to track how many Globs reference a particular VALUEID.) For efficiency, multiple Globs can refer to the same Value. This allows us to use copy-on-write logic; we pass auto variables between functions without copying the Values unless the variables are modified. For example, consider an auto variable pointing to a Glob pointing to a Value that is storing an array. When the variable is passed as an input parameter to a function, a new Glob with a unique GLOBID is allocated and referenced by a new auto variable in the parameter list of the called function. The new Glob begins by pointing to the same Value as the Glob referenced by the caller. Both Globs have a ref-count of one and the Value has a ref-count of two. If the function modifies the array, the Value is copied into a new Value, and the second Glob is modified to point to the new Value, while the ref-count of the old Value is decremented to one. Both Globs still have a ref-count of one, both Values now have a ref-count of one, and the structures are now independent of each other. If the called function does not modify the array, the Value is never copied and the ref-count of the Value is simply decremented when the function returns, saving an expensive copy operation. (When modifying a Value, if more memory is not needed, it should only be copied if its ref-count is greater than one; otherwise it should be modified in-place for efficiency.) The same logic applies to assignment. When one auto variable is assigned to another, a new Glob is allocated, but the Value is reused until either of the variables is modified. The same logic also applies to elements stored in the example array, not just the array itself, and so we can casually pass large complex data structures between functions without an expensive copy operation except when the structure is modified. Even when the structure is modified, usually only part of it needs to be copied. This has the surprising effect that code that uses auto variables to store arrays and hashes may perform substantially faster than code that uses primitive strings, arrays, and structs, depending on how the variables are used (unless of course, similar copy-on-write logic is reimplemented manually).

In the Glob schema, the value can be any single value from several possible sets. As this is the core of the specification, we'll define it first to give the reader a context for understanding the schemas that follow. This schema represents a structure in memory that represents a reference count and a value. Notice that here we say the value element in a Glob can be undef, another GLOBID, or a VALUEID. Undef is represented directly in the glob, as it requires no additional information. One way to implement this, assuming Glob is a struct and value is a pointer in it, would be to set the value element to null to represent undef. A Glob that references another Glob is also directly represented here, rather than as a Value, because again we can simply use the same pointer for referencing a Glob or a Value, and allocating a structure inbetween would just be a waste of memory and CPU time. It is necessary to store what the type of the value is to differentiate between the different types that can be stored in Value and a pointer to a Glob. However, exactly how and where to store the type is not important here; in the specification, we differentiate by checking which set the value is in.

Glob

refcount : $\mathbb{N}$

value : UNDEF $\cup$ GLOBID $\cup$ VALUEID

[VALUEID is explained later]

For convenience in our specification, we'll explicitly track all the Globs using a partial function to map GLOBIDs to Globs. An explicit structure for this is not necessary during implementation, as the actual memory address suffices to uniquely identify a Glob, but here we must use a GLOBID to clarify whether two Globs with the same value are distinct, or whether they are the same Glob with two references. For the specification, we also need this to represent operations to create and destroy Globs. Free Globs always have a refcount of zero and a value of undef. The domain of free_globs represents all the possible return values for the memory allocator.

AllGlobs

allocated_globs : GLOBID ⇸ Glob
free_globs : GLOBID ⇸ Glob

dom allocated_globs ∩ dom free_globs = ∅ [No overlap]
dom allocated_globs ∪ dom free_globs = GLOBID[All are represented]
∀ x : ran allocated_globs • x.refcount > 0
∀ x : ran free_globs • x.refcount = 0 ∧ x.value = undef

The initial state of the system has no allocated globs. Notice that this one initialization schema implies the initial state of the other schemas that follow, such as AllAutos.

AllGlobs_Init

AllGlobs

dom allocated_globs = ∅ [Implies all are in free_globs]

AllAutos is a container representing all the auto variables. This includes both the auto variables created by the runtime environment via code emitted by the compiler when translating the program and auto variables defined by operations in the schemas in this specification, such as the reference operator and storage in arrays and hashes. For example, if an auto variable is storing a reference to an auto variable, the GLOBID for both the reference and the referent are included in the bag called autos. When a function is entered or exited, the code emitted by the compiler inserts code to update this by calling either $lang.auto.new.undef (represented by AllAutos_Create) or $lang.auto.refcount_inc (represented by Auto_IncRefcount), depending on whether it's a new variable or an alias to an existing variable, and $lang.auto.refcount_dec (represented by Auto_DecRefcount). It is up to the compiler to add and subtract the references it creates from the refcounts by calling these functions; this is outside the scope of this specification because it depends heavily on the syntax of the language. $lang.auto.refcount_dec will never be called without a corresponding previous call to $lang.auto.refcount_inc; the compiler guarantees this. (The reverse is possible; $lang.auto.refcount_inc can be called without a corresponding call to $lang.auto.refcount_dec if, for example, there is a function that loops infinitely, preventing the second call from ever executing. This is not a concern.)

```
AllAutos
AllGlobs
autos : bag GLOBID
---------------------------------------------------------------
                                   [ This means a Glob’s refcount must match ]
                                   [ the number of occurances of it in the bag. ]
∀ x : dom autos • autos # x = allocated_globs(x).refcount
                                   [ This means a Glob that’s a reference to a ]
                                   [ Glob must put that reference in the bag. ]
∀ x : { x : ran allocated_globs | x.value ∈ GLOBID } • x.value ∈ autos
```

When a new Glob is created, its ref-count always starts at one and its initial value is always undef. We represent creating Globs as an operation on All_Globs that returns a Glob.

```
AllAutos_Create
ΔAllAutos        [ This operation corresponds to $lang.auto.new.undef ]
glob_id! : GLOBID
---------------------------------------------------------------
glob_id! ∈ dom free_globs
allocated_globs’ = allocated_globs ⊕
   { glob_id! ↦ ⟨ refcount ↦ 1, value ↦ undef ⟩ }
                                   [ This implicitly adds glob_id to the ]
                                   [ autos bag once because refcount is 1. ]
dom free_globs’ = dom free_globs \ glob_id!
```

AllAutos_IncRefcount and AllAutos_DecRefcount implicitly update the ref-count on the corresponding Glob, as described by AllAutos and AllGlobs.

```
AllAutos_IncRefcount
ΔAllAutos        [ This operation corresponds to $lang.auto.refcount_inc ]
glob_id? : GLOBID
---------------------------------------------------------------
autos’ = autos ⊎ ⟦ glob_id? ⟧
```

Notice AllAutos_DecRefcount is a very loaded operation. When it is called, if there are no more occurrences of the glob_id, it implicitly moves the Glob from allocated_globs to free_globs, again because of the constraints in AllAutos and AllGlobs. This, in turn, implicitly sets the value to undef.

AllAutos_DecRefcount

ΔAllAutos [This operation corresponds to $lang.auto.refcount_dec]
glob_id? : GLOBID

autos’ = autos - ⟦ glob_id? ⟧ [Missing symbol for bag-difference.]

5.4 Structure of encapsulated values

The above schemas define our core memory structure and how to track ref-counts. Now we define the various things that can be stored in an auto variable. In the interest of efficiency, we have already defined undef and a reference to another Glob as possible values represented without the infrastructure below. In the implementation, it may be desirable to represent other common cases (such as integers that are no larger than a pointer) directly in the Glob rather than in the separate allocations below; however if this is done, the cases must not require the Glob to be resized, because that will sometimes require the Glob to be moved to a new location in memory, which will break references to it. In short, the Glob’s address must not move after it is allocated, so any dynamically sized parts of a variable must be stored separately using the infrastructure below. This is the main reason that values are not stored directly in Globs.

First, we need an ADT to hold the ref-count. We use a separate schema for this because the ref-count is always there regardless of which type of value is assigned. A suggested implementation is define a struct for each type of value, all of which store the ref-count as the first element. Then when the ref-count is needed, it can be easily accessed without regard for which struct definition this is. To minimize the number of calls to the memory allocator, all the information for a Value should be stored in the same chunk of memory, if possible (hashes are probably the only exception to this).

Value

refcount : ℕ
value : Integer ∪ Fraction ∪ String ∪ Array ∪ Hash ∪ Fref

AllValues is quite similar to AllGlobs; it gives us a way of tracking which Values are unique and it enforces ref-counts. The last part of the predicate in AllValues looks obtuse, but it is just expressing what we already intuitively know ref-counts of values to be. It is saying that the ref-count of a Value is always equal to the number of Globs that refer to its VALUEID.

AllValues

AllGlobs

$\text{allocated\_values} : \text{VALUEID} ⇸ \text{Value}$

$\text{free\_values} : \text{VALUEID} ⇸ \text{Value}$

$\text{dom allocated\_values} \cap \text{dom free\_values} = \varnothing$

$\text{dom allocated\_values} \cup \text{dom free\_values} = \text{VALUEID}$

$\forall\, x : \text{ran allocated\_values} \bullet x.\text{refcount} > 0$

$\forall\, x : \text{ran free\_values} \bullet x.\text{refcount} = 0$

$\forall\, v : \text{dom allocated\_values} \bullet \text{allocated\_values}(v).\text{refcount} =$
$\quad \#\{\, g : \text{dom allocated\_globs} \mid \text{allocated\_globs}(g).\text{value} = v \,\}$

Finally, after much infrastructure, we define the structure of each possible type. These should mostly be fairly obvious; all the difficult parts are expressed above. Integers and fractions are particularly easy possibilities. Notice that there is no bound on integers or on the numerator or denominator for fractions. This means that if an operation's result cannot be stored in the amount of memory available, more must be allocated.

Integer

$\text{value} : \mathbb{Z}$

Fraction

$\text{num} : \mathbb{Z}$

$\text{denom} : \mathbb{N}$ [The denominator must be positive.]

$\text{denom} > 0$

$\text{num} = 0 \Rightarrow \text{denom} = 1$ [Avoid dividing by zero.]

[This says fractions must be stored in reduced form.]

$\text{num} > 0 \Rightarrow \neg \exists\, x : \mathbb{N} \bullet (x > 1) \wedge (\text{num} / x \in \mathbb{N}) \wedge (\text{denom} / x \in \mathbb{N})$

A String is a homogenous sequence of integers. (A primitive "string" can be any homogeneous sequence, like what C calls an array. The only purpose of supporting "string" separately from "array" in the context of auto variables is to provide an acceptably efficient way to represent sequences of integers. Allowing an unrestricted subtype here would require so much complexity it would defeat this purpose.) While usually the integers will be unsigned and 8 bits wide, the implementation must support both signed and unsigned at any size. Any width of integer is valid; even strange widths such as 13-bits must be supported. Also, while the widths of elements in a single string must be constant, the widths of elements may be different widths for different strings in the same program. During storage, the sizes may be increased. For example, a string of 13-bit integers may be represented internally using 16-bit integers for efficiency. However, there must be no way for outside code to tell the difference between an implementation that uses true 13-bit integers and an implementation that uses integers padded to 16 bits. When deciding whether to use padding, the tradeoffs between the cost of conversions and the inefficiency of unaligned integers must be carefully weighed.

String

width : $\mathbb{N}$

value : seq $\mathbb{N}$

signed : boolean

---

width > 0

if signed then

$\forall$ char : value $\bullet$ $-2^{\text{width - 1}}$ <= char < $2^{\text{width - 1}}$

else

$\forall$ char : value $\bullet$ 0 <= char < 2^{width}

An Array is a sequence of auto variables, allowing it to be heterogeneous. The contained auto variables may contain any valid type of data, including, for example, another Array.

Array

value : seq GLOBID

[This means all elements in the Array must]
[must increment the reference count (through]
[the autos bag).]

$\forall$ x : value $\bullet$ x $\in$ autos

A Hash can be thought of as a dynamic representation of a primitive struct. More precisely, it is a mapping of keys (a domain of unique strings, colloquially known as keys) to values (a range). It has all the properties of a partial function: each key must be unique, but a value can occur many times. It also has all the desirable properties of a hash table: finding or deleting a key is an O(1) operation, and inserting a key is an O(log(n)) operation (incurring the log(n) cost because sometimes the table must be extended, which requires reorganizing all the keys). Additionally, however, our Hash object remembers the order in which the keys were inserted (unless the order is explicitly modified), so iterating through the keys always yields the same order of elements. This may be implemented trivially by forming a two-way linked list out of the keys, in addition to inserting the keys into the hash table. There is no cost for this in terms of time-complexity (and only an irrelevant fixed cost to each write operation to maintain a couple of extra pointers), yet it adds much expressive value to the data structure. It also makes Hashes capable of everything a primitive struct is capable of. To achieve this efficiency, it must be a single hybrid data structure, not two separate structures, but in our specification, we represent the ordering of the keys separately from the partial function.

Hash

value : String ⇸ GLOBID
order : seq String

[This means all elements in the Hash must]
[must maintain the reference count (through]
[the autos bag).]

$\forall\, x : \text{ran value} \bullet x \in \text{autos}$
dom value = ran order [Value and order describe the same set.]

It would be infeasible to support auto references to all the various combinations of types that a pointer can reference. However, supporting references to auto variables is necessary to support complex data structures, and supporting references to functions is necessary to support abstract data types. (Further discussion of the implementation of abstract data types is beyond the scope of this discussion.)

In Pit’s terminology, a pointer is just like C’s pointers, which are a primitive integer storing a memory address with an optional static sub-type property that allows it to be dereferenced. Pointer arithmetic and conversions to and from integers are allowed. Pointers cannot reference auto variables. An auto variable cannot store a pointer, except by interpreting it as an integer when stored and providing a typecast when fetched. A reference is an auto variable that refers to another auto variable; it is a type that can only be stored in an auto variable, and it can only refer to auto variables. References do not allow arithmetic or conversions to and from integers. References are strong, meaning that they increment the reference count of the referent, thus preventing garbage collection for the duration of the reference.

Because a reference is just a pointer in the context of an auto variable with certain restrictions, we do not need to allocate an additional structure for it, and formally we have no additional schemas to represent it. As described above in the Glob schema, the value element of a Glob can simply store a GLOBID directly.

Fref means “function reference”. These function references are restricted to functions with a certain prototype by the compiler, so we don’t have to represent the prototype dynamically.

[FUNCTIONID]

Fref

value : FUNCTIONID

5.5 Operations

These schemas represent the various operations that can be performed on auto variables. It is worth clarifying the context of these operations. Remember, we're trying to represent the semantics of operations on auto variables as written in Pit. A schema that represents an operation, for example, Auto_Add, creates a new structure to hold the return value of the operation. For example, the representation of "$a = $b + $c - $d" is, informally, three operations, each requiring a place to hold the return value:

AllAutos_Create(glob_id! ↦ $temp1)
Auto_Add(left ↦ $b, right ↦ $c, result ↦ $temp1)
AllAutos_Create(glob_id! ↦ $temp2)
Auto_Subtract(left ↦ $temp1, right ↦ $d, result ↦ $temp2)
AllAutos_DecRefcount(glob_id ↦ $temp1)
AllAutos_Create(glob_id! ↦ $temp3)
Auto_Assign(left ↦ $a, right ↦ $temp2, result ↦ $temp3)
AllAutos_DecRefcount(glob_id ↦ $temp2)
AllAutos_DecRefcount(glob_id ↦ $temp3)

Obviously, substituting $a for $temp2 and eliminating $temp3 and the Auto_Assign operation, would be more efficient, but we are interested in the general case, not optimized cases, at this point, and in the general case, every operation has a return value. There are several oversights and syntactic problems in the example above, but we'll ignore these for the moment; they are necessary as we map the informal description to the formal specification because for the moment we are not interested in a formal specification of the language's syntax. The return value of the last operation in a statement gets its ref-count decremented or is freed.

Clearly, most of the binary operators will be nearly identical in their specification. They are so similar and there are so many, it would be confusing and a waste of time to specify all of them. We will specify only one operation for each set of operations that are almost identical. For example, subtract, multiply, divide, modulus, as well as bit-wise operations (or, nor, and, nand, xor, nxor, shifting) are omitted. Assignment is quite different, as explained below, so this one is not omitted. Unary operators differ substantially from binary operators, so we include negation.

```
Auto_Add
ΔAllAutos
left? : GLOBID
right? : GLOBID
result? : GLOBID
exception! : EXCEPTION
----------------------------------------
if
   Integer_Add(
      left? ↦ allocated_globs(left?).value
      right? ↦ allocated_globs(right?).value
      result? ↦ allocated_globs’(result?).value
   ) ∨ Fraction_Add(
      left? ↦ allocated_globs(left?).value
      right? ↦ allocated_globs(right?).value
      result? ↦ allocated_globs’(result?).value
   )
then exception! = null
else exception! = type_mismatch                [ result is unchanged ]
```

```
Integer_Add
left? : Integer
right? : Integer
result! : Integer
----------------------------------------
result!.value = left?.value + right?.value
```

```
Fraction_Add
left? : Integer
right? : Integer
result! : Integer
----------------------------------------
result!.num / result!.denom =
   left?.num / left?.denom + right?.num / right?.denom
```

Auto_Assign is another loaded operation, because this is where the ref-counts of Values come into play. In the statement “$a = $b”, the actual value is not copied. As described above, to accomplish this, Value has a ref-count that is distinct from Glob’s ref-count. For example, assume the ref-counts of the Globs for $a and $b start at one; these do not change. The sequential logic of the assignment is as follows:

- Decrement the ref-count on the previous Value of $a (and free it if zero).
- Increment the ref-count of the Value for $b twice.
- Point the Glob for $a at the Value for $b.
- Point the result Glob at the same Value.

After this, $a and $b still have the same distinct Globs, but these Globs point to the same Value. If, for example, $b is incremented later with the statement "$b += 1", the link is broken. A new Value for $b is created, and the Glob for $b is pointed at it. The old Value's ref-count is decremented, but not freed because $a still points to it. Notice that this is not the behavior expected from a reference; if $a were a reference to $b, for example through the statement "$a = $b<-", the ref-count for the Value of $b would be unaffected. In this case, the Glob for $a stores a pointer to the Glob for $b. This time, the ref-count for the Glob for $b, rather than its Value, is incremented. Afterwards, $a and $b still have distinct Globs, but the Glob for $a points to the one for $b.

Auto_Assign

ΔAllAutos
left? : GLOBID
right? : GLOBID
result? : GLOBID
exception! : EXCEPTION

[Implicitly increments refcount of new value]
[and decrements refcount of previous value.]
allocated_globs'(left?).value = allocated_globs(right?).value
[Implicitly increments refcount of new value.]
allocated_globs'(result?).value = allocated_globs(right?).value
exception! = null

The reference operator in the above example "$a = $b<-" is represented by Auto_Ref; it is relatively straight-forward.

```
Auto_Ref
ΔAllAutos
in? : GLOBID
result? : GLOBID
exception! : EXCEPTION
------------------------------------------------------------
                    [ Implicitly increments refcount of the Glob for in. ]
allocated_globs'(result?).value = in?
exception! = null
```

Negation is a fairly simple case. Bitwise-not is nearly identical to negation, so it is omitted.

```
Auto_Negate
ΔAllAutos
in? : GLOBID
result? : GLOBID
exception! : EXCEPTION
------------------------------------------------------------
if
  Integer_Negate(
    in? ↦ allocated_globs(in?).value
    result! ↦ allocated_globs'(result?).value
  ) ∨ Fraction_Negate(
    in? ↦ allocated_globs(in?).value
    result! ↦ allocated_globs'(result?).value
  )
then exception! = null
else exception! = type_mismatch                  [ result is unchanged ]
```

```
Integer_Negate
in? : Integer
result! : Integer
------------------------------------------------------------
result!.ref-count = 1
result!.value = - in.value
```

```
Fraction_Negate
in? : Integer
result! : Integer

result!.num / result!.denom = (- in?.num / in?.denom)
```

6 Conclusion

Syntax and semantics are central to any programming language. When developing a programming language such as Pit, it is quite helpful to specify these constraints formally because creating a compiler is difficult, even with a clear understanding of the language. The design of a language can benefit from the application of formal methods by enhancing understanding and communication between collaborating developers[1]. A formal specification also provides a context in which correctness, an important property of a compiler, can be formally analyzed. Additionally, it helps make the language's description more precise and unambiguous, making it easier for a user to learn. It is also necessary to carefully analyze important aspects of the language during the early stages of development to help avoid inconsistencies. This was the main objective of this effort, and it was successfully achieved using Z notation[5]. Along with these advantages, Z notation was effective in representing the auto variable type.

7 References

[1] Jeannette M. Wing. A specifier's introduction to formal methods. Computer 23, 9 (1990): 8-22.

[2] NASA. Formal methods demonstration project for space applications — phase I case study: space shuttle orbit DAP jet select. JPL Document D-11432, (1993).

[3] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Technical University of Munich, 1997.

[4] J. S. Dong, R. Duke, and G. Rose. An object-oriented approach to the semantics of programming languages. Proceedings of 17th Australian Computer Science Conference (1994): pp. 767-775.

[5] J. M. Spivey. The Z notation: a reference manual. Prentice Hall International (UK) Ltd, 1992. (0-13-978529-9).

[6] Jonathan Jacky, The way of Z: practical programming with formal methods. Cambridge University Press, 1996. (0-521-55976-6).

[7] Dick Grune, C. Jacobs, Koen Langendoen, and Henri Bal. Modern compiler design. Wiley, 2001.

[8] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of art and future direction. ACM Computing Survey 28, 4 (1996): 626-643.

[9] Nathaniel Gray, Cristian Tapus, Aleksey Nogin, and Jason Hicy. Building reliable compilers with a formal methods framework. international symposium on software reliability engineering (2003): 319-320.

[10] M. Mernik and M. Leni and E. Avdi and V. Umer. Reusable object-oriented approach to formal specifications of programming languages. L'object, 4, 3 (1998).

[11] Leif Pedersen. Pit programming language. http://pit.devpit.org/.

[1] @article{102816, author = {Jeannette M. Wing}, title = {A Specifier's Introduction to Formal Methods}, journal = {Computer}, volume = {23}, number = {9}, year = {1990}, issn = {0018-9162}, pages = {8--23}, doi = {http://dx.doi.org/10.1109/2.58215}, publisher = {IEEE Computer Society Press}, address = {Los Alamitos, CA, USA}, }

[2]

[3] @PhDThesis{Poetzsch-Heffter97specification, title = {Specification and Verification of Object-Oriented Programs}, author = {Arnd Poetzsch-Heffter}, school = {Habilitation thesis, Technical University of Munich}, note = {Available}, month = jan, year = 1997, }

[4]

[5] @book{129612, author = {J. M. Spivey}, title = {The Z notation: a reference manual}, year = {1992}, isbn = {0-13-978529-9}, publisher = {Prentice Hall International (UK) Ltd.}, address = {Hertfordshire, UK, UK}, }

[6] @book{249512, author = {Jonathan Jacky}, title = {The way of Z: practical programming with formal methods}, year = {1996}, isbn = {0-521-55976-6}, publisher = {Cambridge University Press}, address = {New York, NY, USA}, }

[7] @book{556249, author = {Dick Grune and C. Jacobs and Koen Langendoen and Henri Bal}, title = {Modern Compiler Design}, year = {2000}, isbn = {0471976970}, publisher = {John Wiley \& Sons, Inc.}, address = {New York, NY, USA}, }

[8] @article{242257, author = {Edmund M. Clarke and Jeannette M. Wing}, title = {Formal methods: state of the art and future directions}, journal = {ACM Comput. Surv.}, volume = {28}, number = {4}, year = {1996}, issn = {0360-0300}, pages = {626--643}, doi = {http://doi.acm.org/10.1145/242223.242257}, publisher = {ACM}, address = {New York, NY, USA}, }

[9] @inproceedings{Gray03buildingreliable, author = {Nathaniel Gray and Cristian Tapus and Aleksey Nogin and Jason Hicy}, title = {Building reliable compilers with a formal methods framework}, booktitle = {The 14th International Symposium on Software Reliability Engineering (ISSRE 2003). Supplementary Proceedings}, year = {2003}, pages = {319--320}, publisher = {Chillarege Press}, }

[10]

[11]