PIT: A NEW LOW-LEVEL LANGUAGE

by

Leif Pedersen
Bachelor of Science, University of North Dakota, 2005

A Thesis
Submitted to the Graduate Faculty

of the

University of North Dakota

in partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota
May
2009

This thesis, submitted by Leif Pedersen in partial fulfillment of the requirements for the Degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

_____
Chairperson

_____

_____

This thesis meets the standards for appearance, conforms to the style and format requirements of the Graduate School of the University of North Dakota, and is hereby approved.

_____
Dean of the Graduate School

_____
Date

PERMISSION

Title Pit: A New Low-Level Language

Department Computer Science

Degree Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the chairperson of the department or the dean of the Graduate School. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

_____

Signature

_____

Date

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

ABSTRACT

Pit is a new language for low-level programming, designed to be a self-hosting alternative to C. The novelty is it supports automated memory management without excluding manual memory management, and without hindering key features associated with low-level programming, such as raw pointers, inline assembly code, and precise control over execution. This paper presents Pit as a language, then it presents several areas of research related to Pit.

The first area of research examines how Pit's approach to memory allocation can be used to significantly increase the security of low-level programs. Automatic memory allocation is a useful tool of abstraction in many situations. Since Pit provides this tool without hindering low-level programming, it allows automated memory management to be used in programs where it previously could not be used, such as kernels. Specifically, this tool of abstraction can assist the programmer significantly in writing low-level code with fewer security problems caused by buffer overrun or integer overflow bugs by reducing the number of opportunities for such bugs in areas of code that do not need the precision of manual memory allocation. Existing solutions, such as Cyclone add various ways of checking bounds, but have two major disadvantages: they require extra work from the programmer, and they detect but do not fix memory allocation bugs. Pit's approach simplifies what the programmer writes, making code more understandable.

The second area of research examines how Pit's approach to memory allocation can be used to make low-level user-land programs more portable. Using more flexible

data types can loosen the bindings between programs and the system libraries at the binary interface level, and define an API to system libraries that does not require object files to be recompiled because of minor implementation details when moving from one operating system to another.

In the third area of research, Pit is a case study for showing how to use Z notation to formally specify semantics of a language. 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.

CHAPTER 1. INTRODUCTION

General-purpose programming languages, referring to languages designed for writing wide ranges of applications such as C, Perl, and their relatives, may be characterized by their style of memory management and level of abstraction. As a general trend, less abstract languages require the programmer to explicitly code memory management logic and statements express CPU instructions more directly with greater detail, while more abstract languages require less attention to memory management and represent the intent of the program more directly with less detail. There are, of course, many points of view on classifying languages, but for this discussion, the interesting trade-offs between general-purpose languages are manual or automatic memory management, representing CPU instructions directly or the intent of the algorithms, and representing more detail or less detail. For example, C represents the CPU's instructions almost directly and provides no automated memory management and requires more detail in the code, however Perl manages memory entirely automatically and usually represents complex algorithms in less code. The cost to using Perl is, of course, efficiency; in many cases, Perl code is so slow it cannot meet the user's requirements. This pattern of trade-offs frequently applies to comparisons between a low-level and a high-level language. This brief discussion may seem to imply a trade-off between automated memory management and expression of fine-grained detail, however even if this connection holds (and it may not) it does not exclude the possibility of creating a language that can express

both sides of the trade-offs in adjacent statements. This duality is precisely what Pit implemented.

Unfortunately, most software does not divide as neatly as most languages into "low-level" and "high-level" where, for example, low-level applications cannot afford the cost incurred by high-level languages and high-level applications cannot afford the cost of extra effort in development time incurred by low-level languages. Most nontrivial software has some code that does not need to execute efficiently and therefore would benefit from a high-level language and some code that must execute quickly or for other reasons cannot afford the costs of a high-level language. Graphical programs are an easy example of this; code that deals with individual pixels and polygons for rendering images and animations benefits significantly from optimizing at the level of individual CPU instructions, while code that directs larger user interface widgets such as window placement and buttons benefits far more from greater abstraction.

It's possible to bridge this gap by using two languages to write a program, but this approach has disadvantages. One problem is it's difficult for a module written in one language to access data stored in memory by a module written in a different language because most languages store data in memory quite differently from each other. Perhaps more importantly, using two languages to implement the software creates an artificial boundary in the logic based on the capabilities of the two languages rather than based on a natural separation of concern that's convenient for the program's design. A better solution is to use a language that allows both approaches and allows fine-grained control over when each approach is applied. Pit uses variable declarations to determine which approach to use, which allows convenient yet precise control.

Pit[15][16] is a compiled language derived primarily from C. While it is designed to be familiar to C programmers, the languages are different enough in coding style that it will probably always be a manual task to translate between C code and Pit code in a way that produces reasonably understandable code. The novelty is that it simultaneously supports the variable types found in C, called "primitive variables" and another class of variables called "auto variables". Auto variables are declared syntactically using the keyword "auto" in place of the type specification. Auto variables are similar in many respects to variables implemented by Perl; in this regard, Pit is secondarily derived from Perl. They are dynamically typed and manage memory automatically. The programmer can choose to write the entire program using only auto variables, only primitive variables, or a combination of both. This allows the programmer to choose to use automated or manual memory management with every variable declaration. Thus, the programmer can write some code that has all the characteristics of C code by avoiding auto variables, and in the same function write more code that has all the benefits of automated memory management and dynamically typed variables by using auto variables.

Pit (named simply for Devpit.org) began as a classroom project to implement a C library for implementing dynamically-typed variables in C, which eventually developed into Pit's concept of auto variables. Its API used a single struct pointer type for variables which could store any type of value. The struct was opaque to the programmer, but indicated whether it contained an integer, string, hash, etc. We analogized this variable type to a Perl scalar variable. The API provided functions for the programmer to call for each operation, such as assign, add, reference, dereference, etc. Much like with Perl scalars, each variable carried a reference counter, which, for example, incremented when

storing the variable in a larger structure. The programmer had the responsibility of calling a function which would decrement this reference counter or free the variable (recursively for multilevel data structures) every place execution could exit the scope containing the variable. This wasn't bad as far as complex data structure manipulation in C goes, but it was far too clumsy for managing simple strings; nobody writing "real" software would want to use this for simple things. We finally redesigned this concept as a new language. This has several advantages. Most importantly, it simplifies the concept of auto variables enough that they are easier to use than primitive variables, which are still as easy to use as C's variables. now the compiler automatically inserts the calls to decrement the reference count or free these variables; this tips the balance, making it easier to use than manual allocation. Further, we could now add several important features missing from other low-level languages such as exceptions and namespaces. Also, the programmer can write operations on the new variable type with symbolic operators in the same way as traditional variables, rather than as function calls. In adding these features, we kept the language within constraints that make suitable for the lowest-level programs traditionally written in C, especially kernels. To be sure, other languages have these features, but none of them fit within these constraints. During this work, we studied ways that these improvements could be used for better security, portability, general maintainability, etc in low-level programs.

The compiler implements auto variables by inserting function calls to a supporting library. Each operation translates into a function call, and it inserts additional calls for incrementing and decrementing reference counts for garbage collection. It also transparently inserts calls to this library for converting from a primitive variable type to

an auto variable or vice versa, making it easy to assign a value from a primitive variable to an auto variable and vice versa. Pit is self-hosting, meaning that it can express its own compiler tools and supporting libraries with no tools written in other languages (although this work is in progress). To accomplish this, the supporting language library uses primitive variables to construct the data structures that store information for auto variables.

Pit is designed to be suitable for (but not limited to) low-level programming, such as for operating system kernels, device drivers, system libraries, etc. This is perhaps the class of software that is most starved for improvements in languages because most implementations of recent languages require a virtual machine, prevent manual memory management, and don't have a way to access assembler code. Pit can interact with C code reasonably easily, which makes transitioning or interacting with an existing project easier. All C variables map directly to primitive variables in Pit, and the calling convention for functions is close enough that a simple wrapper function can work around the differences.

CHAPTER 2. EXAMPLES OF PIT SYNTAX AND TRANSITION 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.

```
1.  ...
2.  private array(struct("key0" array(char, 64), "key1" array(char, 64)), 2) $root;
3.  $root(0)("key0") = "array element 0, hash element key0";
4.  $root(0)("key1") = "array element 0, hash element key1";
5.  $root(1)("key0") = "array element 1, hash element key0";
6.  $root(1)("key1") = "array element 1, hash element key1";
7.  ...
```

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:

```
1.  // "$root" above is translated into "%base - 64b" here, which was already
2.  // allocated in the stack frame.  After each group of statements, the
3.  // stack is back to its initial state.
4.
5.  // Translation of: private auto $root;
6.  stack_alloc 64b;                     // Subtract 8 bytes (64 bits) from
7.                                       // the stack pointer
8.  call $lang.auto.new.undef;           // Constructs a new auto with the
9.                                       // value undef and ref-count of 1
10. store <ptr> [ %base - 64b ], <ptr>;  // Pops the pointer to the new auto
11.                                      // and stores it at %base - 64b
12.
13. // Translation of: $root = array{};
14. stack_alloc 64b;
15. call $lang.auto.new.array;           // Constructs a new auto with the
16.                                      // value of an empty array
17. store <ptr>, <ptr> [ %base - 64b ];  // Pushes the location of $root
18. call $lang.auto.refcount_inc;        // Inc ref-count because a binary
19.                                      // operator call decrements the
20.                                      // ref-count of both inputs
21. call $lang.auto.binop.assign;        // Perform the assignment, store the
22.                                      // result (a temp auto holding the
```

```
23.                                     // return value of the assignment)
24.                                     // in the second parameter's position
25. stack_free 64b;                     // Discard pointer to $root
26. call $lang.auto.refcount_dec;       // Free the temp auto
27. 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.

CHAPTER 3. COMPARING PIT TO OTHER LANGUAGES

Table 1. Compare major features of Pit, C, and Perl. Well-supported features are marked Yes, missing features are marked No, features that are so clumsy they are seldom used are marked Clumsy. Perl's design entirely obviates separate module interfaces, so this is marked Not Applicable. Further explanation for each row is in the section referenced in the last column.

Feature	Pit	C	Perl	See section
Basic features				
Manual memory allocation	Y	Y	N	3.1
Automatic memory allocation and GC	Y	N	Y	3.1
Automatic module interfaces	Y	N	NA	3.2
Namespaces	Y	N	Y	3.3
Preprocessor	N	Y	N	3.3
Language constructs				
Homogeneous arrays	Y	Y	C	3.4
Heterogeneous arrays	Y	N	Y	3.4
Structs	Y	Y	N	3.5
Hashes	Y	N	Y	3.5
In-line assembly	Y	Y	N	3.6
Exceptions	Y	C	C	3.7
High-level features				
Abstract data types	Y	N	Y	3.8
Automated compilation	Y	N	Y	3.3
Native vector math	Y	N	N	3.9
Eval	N	N	Y	3.10

The first step to grasp how a new language fits into everyday programming is to gain an intuitive understanding of how design patterns for programs written in the language differ from those written in other languages, especially the languages that are most similar and influential to its design. A good next step is to examine how programs

will be simpler or more complex because of the capabilities and limitations of the language. This helps present the language in the context of practical use.

Comparing Pit to C and Perl is useful because Pit is designed to make C's capabilities and Perl's capabilities easy to use together. This will highlight Pit's additional capabilities, acknowledge its limitations, and provide insight into what Pit code looks like and ways it may reduce the effort required to write code. Table 1 shows an overview of which common features are supported in Pit, as compared with C and Perl. The following sections describe each feature in more detail.

3.1. Manual and automatic memory allocation

This feature is, of course, the focus of this paper. In Pit, the programmer chooses which variables are allocated automatically. If the programmer declares a variable to be the auto type, the memory for it and anything stored in it is allocated automatically, and reference-counting garbage collection is performed automatically, similarly to Perl. If the programmer declares a variable to be a primitive type, he must manually allocate and free memory the same way he would in C. The language is designed to generate code that is predictable and consistent in terms of CPU usage and memory usage; this is the primary reason it uses reference-counting garbage collection instead of mark-and-sweep garbage collection.

3.2. Automatic module interfaces

In C, the programmer must meticulously create a header file for every module. Typically, almost all the information in the header file is duplicated in the module's source file. All of the duplicate information could be generated automatically, and

therefore almost all of the time programmers spend maintaining header files for C modules is wasted.

Perl has an unusual approach. When a module is imported, it is literally executed. This is because, roughly speaking, Perl code is compiled at run-time, and declaration statements are “executed”. In any case, the net effect is that it supports modules in a conventional sense, but does not require manual interface specifications.

Pit uses “interface files”; when a module is compiled, the compiler generates both the object file containing machine code and an interface file for the module. The contents of the interface file fulfill the same role as the contents of most C header files; it contains only the prototypes for functions and shared variables. In contrast to C, however, the compiler generates them automatically. When distributing a module, the interface file and object file may be distributed without the source file.

3.3. Namespaces and automated compilation

C simply does not support namespaces, although most of its successors do. Perl supports namespaces quite well in the form of what it calls “packages”. Pit supports namespaces also, including support for declaring “private” symbols that are only visible inside the namespace and “public” symbols that are visible everywhere. References to public symbols must include the prefix of the name of their namespace except when they are referred to within their own namespace. Pit implicitly declares a namespace for each module with the name of the file containing it; this aids automated compilation, described below. Namespaces may be nested. A file system hierarchy may represent nested namespaces surrounding individual modules, and a module may contain multiple namespaces within the namespace defined by its filename.

When compiling a C program, the programmer must somehow describe each command to execute. The compiler must be invoked for each module of the program. Then the modules must be linked with the necessary libraries to produce the final executable file. Little of this process is automated or standardized. While tools such as “make” ease this process, it is still a complex process that the programmer must design carefully. Most other languages have a better system.

When compiling a Perl program, compilation starts with the main module. When a module requires another module, it contains the “use” statement, which tells the compiler the file name of the required module. The compiler then takes a detour to compile that module before continuing. This process eliminates header files and makes it easier to move the program between systems, since the process of finding a dependency is contained in the compiler rather than, for example, in a Makefile included with the program’s source code.

Pit follows an approach similar to Perl. Because the compiler automatically declares a namespace for each module, and each symbol is prefixed with the name of its namespace, the compiler can find the source file that contains each symbol. To start compiling, the programmer executes the compiler and specifies the main module. When the compiler encounters an “import” statement, it searches the file system for the interface file. If it cannot find one or if it finds one that is out of date, it searches for a Pit source file, generates the interface file from it, and adds it to a list of modules that need to be compiled. After compiling the main module, it compiles the modules in that list following the same scheme recursively. When it finishes, it runs the linker and passes it the name of the main module. Similarly to the compiler, the linker finds object files and

libraries on the file system automatically based on the names of the symbols it needs to resolve. This way the system can produce the executable with no manual specification of what commands to run or what source files (besides the main module) to use for input. The language and system libraries and third-party libraries might include the interface and object files without source code. Note that this is okay, because if the interface file and object file for a module are available, the compiler skips compiling that module.

Other languages support similar functionality, but this is a new feature in the realm of operating systems written in C. While other preliminary steps may be necessary to preprocess source code or installation steps may be necessary to copy the program and accompanying files into place, the basic compilation process is automatic and simple.

3.4. Arrays

Typically, homogeneous arrays are less flexible, but they require less memory because the size and type of each element need only be stored once instead of for each element. C supports homogeneous arrays of any type, but not heterogeneous arrays.

Perl only supports homogeneous arrays of 8-bit integers, as a special case for storing text strings efficiently. Surprisingly, this works well for most applications because the majority of homogeneous arrays can either be treated as atomic 8-bit strings or store elements large enough that the extra information does not have a significant impact on memory usage. However, there are important exceptions; for example, Perl cannot deal with a large array of 16-bit integers or floating-point numbers efficiently, making it unsuitable for rendering graphics. (Perl can still play a major roll in a graphics application by using C libraries to deal with image manipulation or 3D calculations. However, coupling two languages is cumbersome.)

Like Perl, Pit's auto variables cannot store homogeneous arrays, except for strings of 8-bit integers (to accommodate UTF-8 strings). Later work will probably improve auto variables to support homogeneous arrays with any subtype. Pit's primitive arrays, like C, support any primitive subtype. Converting between primitive and auto variables is as easy as assigning one to the other, however if an auto variable containing an array is assigned to a primitive array and there is an element of the wrong type or the primitive array is too small, an exception is thrown. If the string is too large, the extra elements are assigned a null value.

3.5. Structs and hashes

Structs are, of course, borrowed from C. A struct is exactly the same thing in C and Pit. Perl has no support for proper structs, but a struct can be approximated with a hash, since structs and hashes represent nearly the same thing: A set of names mapped to a set of values. The primary difference between a struct and a hash is that the elements of a struct are static, while elements of a hash can be added or deleted at run-time. In Pit, auto variables cannot store structs; instead, a struct can be approximated with a hash. Primitive variables can be declared with the struct type.

3.6. In-line assembly

In low-level programming, it is often unavoidable to use in-line assembly code (however undesirable it may be) for interaction with hardware, manually optimizing code, switching tasks or managing memory pages in an operating system, etc. Pit's interface to assembly code is similar to GCC's interface for C.

3.7. Exceptions

Fundamentally, an “exception” is a branch that can exit several functions simultaneously, usually used for error handling. In most languages, exceptions have additional features that significantly improve their usability, but this is the basic meaning. The best that C has to offer are the setjmp and longjmp symbols. These are quite clumsy, as they have no mechanism for allowing intermediate functions to clean up when an exception occurs, and they require a storage variable to be manually passed to functions that may throw the exception.

Perl supports exceptions also, but its support is clumsy in a different way; error messages in Perl are always in the form of user-friendly strings, for example, “No such file or directory”. This means that the only way for a program to trap a particular error is to match the user-friendly string. If the string changes, for example because the user speaks a different language or uses a different operating system, the trap for that particular error breaks. Perl has some libraries to mitigate this problem, but they are not widely used. Pit provides a way to report a dependable identifier for an error, a way to map that identifier to a user-friendly message, and a way for intermediate functions to clean up their workspace when an exception occurs.

3.8. Abstract data types

C can handle abstract data types in an extremely manual way. Perl supports abstract data types quite well. Pit uses auto variables to support abstract data types in a way similar to Perl. However, the description of the mechanics is too lengthy for this paper.

3.9. Native vector math

In Pit, arithmetic operations on arrays perform a reasonable vector operation when appropriate. For example, adding two arrays together produces a third array where each element is the sum of the elements in the same position in the input arrays. Assigning one array to another copies the values as if each individual element were assigned. Adding a scalar to an array adds the scalar to each element in the array. Multiplication and the other arithmetic operators work the same as addition. The benefit of this, besides easy vector math, is that the compiler can easily optimize these operations for the target architecture using the CPU's vector math instructions. In C, optimizing vector math typically requires rewriting code to use architecture-specific libraries.

Unlike C, Pit does not use the name of an array to represent a pointer to the array (rather, it uses the reference operator just like with any other variable). This alleviates the ambiguity as to whether adding two arrays means adding their memory addresses or adding their elements. To do the former, the reference operator is used to get their addresses before the operation.

3.10. Eval

As a compiled language, dynamic code evaluation is difficult, at best. While it is possible to add this later, no work has been done on this. The language itself does not preclude an eval statement; it is only difficult because the compiler for this language generates stand-alone machine code.

Parsing code during run-time is easy; this requires a library that contains most of the logic of the compiler's phases, such as parsing, optimizing, and code generation. When the compiler encounters an eval statement, it should insert calls to the compiler

library. The difficult part is resolving symbol names inside the eval that refer to lexically scoped symbols outside the eval. Currently, the names of these symbols are lost in translation. The compiler needs to be modified to include a table of local variable names in the machine code's output, and there needs to be an API for communicating variable locations and execution results between the eval'd code and the compiled code. This is a brief analysis, so there may be other difficulties to overcome.

CHAPTER 4. IMPROVING SECURITY

Automated memory management typically found in more abstract languages has benefits beyond saving significant amounts of work. It can entirely prevent some classes of security holes, such as buffer overrun[2] or integer overflow[5][1] bugs.

4.1. Related work

General concern over computer security is probably as old as wide-spread use of computers for routine tasks. In 1972 James Anderson published a report for the Electronic Systems Division of the US Air Force[3]. The primary concerns of his report were security between users of time-sharing systems and security between computers on a network. For its age, his report is surprisingly relevant to modern computing. Security between users is at least as relevant today as it was in 1972 (even if multiuser machinery has perhaps declined), because most operating systems use it to minimize the threat that applications pose to each other. For example, a server running Apache and MySQL typically executes them with limited privileges as separate users so that if Apache is compromised, MySQL is still protected. The importance of security between computers has grown explosively as the Internet has grown, because nearly all computers are now connected to the Internet. However, it wasn't until the 1990s that the industry widely acknowledged that the numerous subtle security flaws in software affect everyone, not just organizations with especially sensitive secrets such as the US Air Force. Today, security is such a high priority that many users will accept severe performance penalties (especially in situations where faster hardware can compensate) or use software that is

significantly inferior in other ways. On the other hand, there is significant successful work on reducing the cost of security, where cost refers to every trade-off – performance, maintainability, capabilities, hardware, etc.

Solutions for minimizing security holes in low-level software are still relatively sparse, and many common languages and programs predate wide-spread concerns about security. It wasn't until 1988 that the first documented exploitation of a buffer overrun appeared in the form of the Morris Worm. Interestingly, the report from Cornell on this incident[10] mentions mutual trust between computer users up to this point, and a desire not to "build walls as high as the sky" to protect against intruders. In today's environment of persistent identity thieves, spammers, bot-nets, pirates, and costly hoaxes it seems strange to look back on a time when we were so unconcerned with security. Looking back, it seems obvious that the Morris Worm was a disaster waiting to happen, however at a time before online shopping and personal computers for everyone raised the stakes, it probably wasn't unreasonable to assume attackers could be tracked down and prosecuted the same way they are with physical security breaches.

In 1996, the community's equilibrium was again punctuated by Aleph One. This time, there was no turning back. Computers had permeated the public to the point where too many people were interested in breaking security to ignore the issue any longer, and now Smashing The Stack For Fun And Profit[2] detailed exactly how to take advantage of unchecked buffers with an explanation most novice hackers could understand. This is roughly when intense work started in securing software against dedicated attackers and setting up ways of quickly responding as new security holes are discovered. The pattern of the buffer overrun programming bug has been astonishingly pervasive and persistent.

In spite of 12 years of auditing since Aleph One threw open the gates, old buffer overrun bugs are still frequently discovered, and new ones are frequently introduced. Simply learning to write bug-free code has proven to be no more effective at eliminating buffer overrun bugs than at eliminating bugs in general.

Starting roughly 2002[5][1], attackers started exploiting integer overflow bugs and soon other non-control-data attacks[6]. Exploiting these bugs is a more subtle exercise and varies more between programs, but as it becomes harder for attackers to find buffer overrun bugs, they will undoubtedly focus more on other bugs.

Many tools have been developed that focus on detecting buffer overrun bugs at run-time. These can help developers fix old code and mitigate security risks, but none of the run-time tools can detect a bug without a test case that triggers it, and several types of errors are not reported by any of them, including integer overflow bugs and buffer overrun bugs within a data structure. Dynamic Buffer Overflow Detection[19] surveys some tools in this category including Valgrind[25], CCured[24], CRED[27], ProPolice[29], StackGuard[7], TinyCC[30], Chaperon, and Insure++. Safe Systems Programming Languages[12] surveys SafeC[28], CCured[24], Vault[26][31], and Cyclone[23].

Cyclone[23] represents good work, but does not provide a complete solution. It extends C by tagging pointers syntactically in different ways to provide bounds-checking. However, to take full advantage of its features requires modifications to the code that are so extensive that it is no harder to simply rewrite the code in a new language. Also, the programmer still has to specify how much memory is required, and Cyclone aborts the program when an allocation bug is detected rather abstracting out memory allocation entirely. This protects the system from intrusion, but does not actually fix any bugs.

Often, aborting execution is not an acceptable solution. Further, Cyclone does not address integer overflow bugs. Writing code in Cyclone is more confusing than writing code in C because it provides several tags for pointers and the programmer needs to know which tag to use in each situation. Writing code in Pit with auto variables is simpler and easier, and actually fixes these bugs. Cyclone provides optional garbage collection, but it uses mark-and-sweep techniques, which exclude many important programs including kernel code and time-critical code. (Pit uses reference-counting garbage collection.) In Cyclone, the programmer must still explicitly calculate the amount of memory required for allocations; automating this is a key ingredient of automated memory allocation.

SafeC[28] similarly protects against buffer overruns with run-time bounds checking, but sacrifices performance to avoid code modifications. Vault[26][31] is an entirely new language that does not support pointer arithmetic, which makes it unsuitable for kernel code. It also suffers from the disadvantage of requiring the programmer to explicitly track memory allocations while restricting pointers so tightly that it could have automated memory management entirely.

SmashGuard[14] presents a solution that is interesting because it is implemented in hardware so it requires no modification to application code and incurs no overhead. Its disadvantages are it protects only against attacks that target function call return addresses, it requires minor kernel modifications, and of course, requires new hardware.

Transparent Runtime Randomization[17] and Address Obfuscation[4] change the memory layout of applications with each execution. This is effective against most attacks because the attacker generally cannot predict the behavior of the program closely enough to compromise it. However, this does not protect against non-control-data attacks or

potential denial-of-service attacks because of program crashes. PointGuard[8] is a tool with a different approach but similar results. Rather than rearranging the application's memory layout, it encrypts pointers when they are in memory so that a corrupted pointer will be decrypted to an unpredictable address before being accessed.

The methods that detect an attack in progress and abort the program do not fix the underlying problem that there is a bug in the software. Detection is a big improvement, and in most programs, this is a "safe" thing to do; although the program behaves in an undesirable way, disaster is averted. In many cases, however, aborting a program is dangerous. For example, an abort in the kernel will crash the computer, resulting in down-time and often lost data. A trapped bug in a database server can cause equally catastrophic results. An attacker may exploit any such bug to deliberately keep a service off-line. While sensitive data is not revealed, the solution is still not good enough.

Vault and Cyclone extend C but make code more complicated, and more complication typically means more errors. It is more desirable to simplify code through abstraction instead, and automated memory management is a powerful tool for this. Automated memory management also addresses non-control-data attacks and integer overflows, which most of the above methods do not consider. However, low-level developers have historically avoided languages that provide automated memory management because these languages also prevent access to raw pointers, in-line assembly, and necessary optimization techniques. (Automated memory allocation includes allocating and freeing memory without explicit statements. It also includes extending buffers to avoid overruns and widening integers to avoid overflows.)

This history suggests the following conclusions. C was designed before security was a major concern, and unfortunately in several ways it is impossible to extend C in ways that would solve the problem once and for all without significant changes to all the code in need of protection. The best way to solve memory allocation bugs is obviously to automate memory allocation, but before low-level developers can use automated memory allocation, there needs to be a language that supports both memory allocation and access to raw pointers. Such a language not only fixes the allocation bugs but also simplifies much code.

The primary disadvantage to creating a new language is that it requires rewriting code, but no solution has been presented yet that can fully protect a program without either unacceptable complexity, unacceptable cost to efficiency, or loss of access to raw pointers. Full protection includes protecting buffers that are part of a struct, and by design, C allows an array contained in a struct to be overrun, and many programs utilize that feature in reasonable ways.

4.2. Analysis

Even today, after so much study and work, cutting edge operating systems still have a dismal outlook on security. Even if all the bugs were fixed in existing code, new ones are constantly introduced with new code. For example, nobody reasonably expects the existing operating systems to ever be free of buffer overrun bugs until there are new innovations. Until then, the industry must settle for staying one step ahead of attackers. Operating system code is perhaps the most widely used code, since every computer needs it. This probably makes it the most targeted code for malicious attackers and probably the most widely audited code as well, but to date, nobody has produced a modern operating

system that is free of security problems. OpenBSD is probably the operating system with the most proactive approach to security, and after many years it is still unusual for them to go even a few months without finding another buffer overrun bug[13]. Many good ideas have been developed to improve security for low-level code, and yet all have fallen short of complete success, where a successful approach must achieve a high level of confidence that the code has no buffer overrun bugs, no integer overflow bugs, and do so without a loss of efficiency. In general, when this paper refers to security, the scope is limited to memory allocation, buffer overrun, and integer overflow bugs because they are the bugs that this programming language can help prevent, and this class of bugs covers the overwhelming majority of security holes.[18][9][8].

C is the language of choice for operating system code for good reasons. Low-level programmers absolutely need easy access to in-line assembly code for hardware IO and setting special-purpose CPU registers. Also, they need to be able to write efficient code, as speed is a major point of competition between operating systems; this means that if there are any inefficient high-level features, they need to be optional. (C has no such features) A language that requires an interpreter or virtual machine is clearly not an option. Perhaps most importantly, programmers need to have a clear idea what the output of the compiler will be, because they are working in an environment where many CPU facilities are often unavailable. When the translation is too abstract or layered too deeply, a low-level programmer will often become frustrated because the compiler will generate unexpected output that may, for example, interfere with the kernel's memory management, use floating-point facilities when they are not available, or attempt to access variables that are temporarily unavailable. Complex tools lead to more of these

problems than simple tools of equal power. To address this, source code that does not use auto variables will never be translated into machine code that automatically manages memory. Because the extra features have no impact on the output unless they are used, Pit works at least as well as C for low-level code. Pit fits all of these requirements, and provides a way to choose when to use automatic memory management as well. Pit also interacts more easily with C than most other languages. This makes it a good step forward from C with little cost.

Pit is by no means a silver bullet. The advantage is that the programmer gets to choose whether the benefit is worth the penalty, rather than being forced to write an entire module in a low-maintenance high-overhead language such as Perl or in a high-maintenance low-overhead language such as C. To effectively use Pit, a programmer must choose wisely between primitive and auto variables. This requires a balance of efficiency and maintainability. As long as a programmer only uses auto variables, there is an absolute guarantee that his code will not have these security bugs. However, this guarantee comes at a cost to efficiency that is often significant. On the other hand, every time a programmer uses primitive variables, he runs the risk of introducing security bugs. It is well-established that program execution time is related to Zipf's Law. Specifically, a small amount of code accounts for most of the execution time of a program; a widely accepted estimate is that 10-20% of the code accounts for 80-90% of the execution time. (The sections of code that account for most of the execution time are the "time-critical sections" sections of code.) On the other hand, Zipf's Law does not apply to security. When considering execution time, a program can be somewhat less efficient, but a program cannot be somewhat less secure. It's either secure or not, because when an

attacker discovers a security hole, typically the whole system is compromised. This leads to the conclusion that while all the code in a program must be secure, only about 10-20% of it needs to be efficient. It is easy to take advantage this in Pit. If only 10-20% of the code is written using primitive variables, then the risk of security holes is eliminated in 80-90% of the code. Additionally, this leaves programmers more time to carefully audit the remaining code for security bugs, both because they don't need to audit the other code for these bugs and because the other code was easier and faster to write initially. To benefit from this aspect of Pit, a programmer must to be willing to accept this trade-off and use it to his advantage. Most of the existing research for detecting security errors in C code or aborting a C program when a buffer overrun occurs is easy to adapt to Pit's primitive variables. Such an adaptation should typically assume auto variables have no such security errors and do the security checking only on primitive variables.

Perl has already proven this approach. When using Perl, programmers write most of the code in Perl, but call C libraries to do the small amount of work that consumes a large amount of CPU time. The Perl code cannot have buffer overrun bugs, (with the bigint extension) integer overflows, double-free errors, memory leaks, non-control-data attacks[6], etc. However, Perl's approach requires the awkward division of a program into high-level and low-level parts and is entirely unusable for writing low-level software, as discussed in the introduction.

4.3. Buffer overrun: C comparison

Simplifying code reduces human error. Human error is the only reason new buffer overrun bugs are introduced into code. We will use a short program written in C and a short program written in Pit to demonstrate how using Pit's auto variables

simplifies code and eliminates the risk of buffer overruns by abstracting the logic for memory allocation out of the written code into the compiler and language library.

Figure 1 shows a C program that adds pairs of numbers read from standard input. Figure 2 shows the same program written in Pit. Notice that in Pit, auto variables can manage all the array sizes and memory allocation, thus eliminating any potential for mismanaged string buffers. As shown in Figure 3, they can also be managed manually just as easily as in C, with the same risk to security. This is the trade-off between improving efficiency and improving security and maintainability.

While Figure 1 has no buffer overrun vulnerabilities, there are several ways they could easily be introduced either when the code is modified or as a simple miscalculation when writing it in the first place. For example, the wrong size could be passed to fgets(), a pointer error could be introduced where the string is parsed with strsep(), or an error could be introduced in the format string passed to printf(). Also notice that the code sets an arbitrary limit on the length of an input line, and while it is obviously possible to work around this, doing so makes the program much more complex and adds more opportunities for security holes. For example, dynamically allocating just enough memory to hold the input can lead to accidentally freeing a dynamically allocated chunk of memory twice[6] or misusing realloc(). These are the very errors that are commonly found in C code, but never found in Perl code.

Figure 2 shows the same program, this time written in Pit using auto variables. Notice that neither the length of a string nor a pointer into the string are ever used, and the bounds on any indexing operations for auto variables are checked at runtime. All the

sizes calculated automatically because memory is managed automatically. This means the opportunities for security holes that were present in Figure 1 are eliminated.

Figure 3 is an exercise in optimization, trading confidence in security for efficiency. It shows the same program written in Pit, this time using some primitive variables. Notice how primitive and auto variables easily work together, allowing the programmer to choose precisely what code to optimize for efficiency at the risk of security and maintainability, and what code to optimize for security and maintainability at the cost of efficiency. In particular, note that $in is implicitly converted to an auto variable when passed to $string.split with little effort on the programmer's part. This level of interaction is impossible when using two languages such as Perl and C together; instead, the programmer must write two separate modules and tie separate functions together. A programmer can use this, for example, to efficiently do time-intensive calculations of plotting a temperature graph from trustworthy input data in the same function that parses distrusted inputs strings from a web browser to choose color preferences or add decoration.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  int main() {
5.      char in[1024], *inptr, *num;
6.      int first, second;
7.
8.      while(fgets(in, sizeof(in), stdin)) {
9.              inptr = in;
10.
11.             if(! (num = strsep(&inptr, " ")))
12.                     continue;
13.             first = atoi(num);
14.
15.             if(! (num = strsep(&inptr, " ")))
16.                     continue;
17.             second = atoi(num);
18.
19.             printf("Sum is %i\n", first + second);
20.     }
21.     return 0;
22. }
```

Figure 1. Example program in C.

```
1.  import io;
2.  import string;
3.
4.  public function(out int $exit_status) $.main = {
5.      private auto $in, $nums;
6.
7.      // $io.stdin is a global symbol.  => is
8.      // the method-call operator.  readline
9.      // is the name of the method that reads
10.     // a line of input.  The line is stored
11.     // in $in and returned.  Dots are
12.     // simply part of a symbol's name
13.     // indicating its namespace.
14.     while($io.stdin=>readline(ret $in)) {
15.             // $string.split() is a function that
16.             // splits $in on " " and stuffs the
17.             // result into $nums as an array of
18.             // strings.
19.             $string.split($in, " ", $nums);
20.
21.             // Convert the strings to integers.
22.             $string.2int($nums(0));
23.             $string.2int($nums(1));
24.
25.             // The object $io.stdout has a method
26.             // called writef that is semantically
27.             // analogous to C's printf().
28.             $io.stdout=>writef(
29.                     "Sum is $sum\n",
30.                     hash {
31.                             "sum", $nums(0) + $nums(1),
32.                     },
33.             );
34.     }
35.
36.     $exit_status = 0;
37. }
```

Figure 2. Rewrite of Figure 1 in Pit.

```
1.  import io;
2.  import string;
3.
4.  public function(out int $exit_status) $.main = {
5.      private string(char, 1024) $in;
6.      private int $in_len;
7.      private auto $nums;
8.
9.      loop: {
10.             // Pit strings are _not_ null-
11.             // terminated.  $in_len is an in-out
12.             // parameter.  Going in it's the
13.             // maximum length of $in and coming out
14.             // it's the actual length of $in.
15.             $in_len = sizeof($in);
16.
17.             // _p means primitive.  => is the
18.             // method-call operator.
19.             $io.stdin=>readline_p($in, $in_len);
20.
21.             // readline_p modified $in_len
22.             unless($in_len) break;
23.
24.             // Split works with autos, so this
25.             // type-cast is required to tell the
26.             // compiler that the string $in is
27.             // shorter than 1024 bytes.
28.             $string.split($in[string(char, $in_len)], " ", $nums);
29.
30.             // Convert the strings to integers.
31.             $string.2int($nums(0));
32.             $string.2int($nums(1));
33.
34.             $io.stdout=>writef(
35.                     "Sum is $sum\n",
36.                     hash {
37.                             "sum", $nums(0) + $nums(1),
38.                     },
39.             );
40.     }
41.
42.     $exit_status = 0;
43. }
```

Figure 3. Some speed optimizations to Figure 2.

4.4. Integer overflow

Integer overflow bugs are more insidious and are a much newer problem than buffer overrun bugs that have not gained notice until around 2002. The pattern of this bug is that the program will behave unexpectedly when an integer's value overflows during arithmetic. These overflows happen in two common situations, either while adding two positive or two negative numbers (or subtracting equivalently) or while

multiplying. When adding a positive number to a negative number there is no problem, and when dividing there is no problem. (Code that calculates powers for memory allocation is much more rare, however that does not entirely exempt power calculations from concern. Multiplication is bad enough for our discussion.)

First, consider adding. Adding two positive numbers with an overflow produces a negative result. An easy example is overflowing while incrementing. While iterating over the characters in a buffer, if the counter variable overflows, the code will accidentally follow a negative offset into the buffer. This particular situation isn't terribly difficult to handle in a number of ways, but extending the example into multiplication shows a far more severe problem.

The resulting sign of a multiplication is not a reliable indicator of overflow. Multiplying two fairly small positive values may produce an overflow so large that the result wraps around more than once, producing a positive result. For example, multiplying 65536 by 65537 using 32-bit integers results in a positive value much smaller than expected. Multiplication is used in calculating the memory address of array elements. This is the root of the problem in the following example.

Just last year (in 2008), CUPS[20] announced a patch[21] for an integer-overflow bug leading to remote exploitation and local privilege escalation[22]. The problem was that the result of multiplying two integers and passing the result to calloc() could result in an allocation smaller than expected. In subsequent code, iterating over this allocation results in overrunning the allocation even though the counter variable stays within reasonable boundaries. If calloc() instead used an unbounded integer for its parameter there is no concern at all. This would allow calloc() to simply return an error indicating

that it cannot allocate the requested amount of memory. The solution for this C code is an awkward check to be sure each input variable is not greater than 32767. Additionally, there is nothing to indicate to code maintainers that part of the reason for this bound is the size of the type lchar_t. Note that this class of problems and this solution can be extended to most functions that write to caller-allocated buffers, such as read(), fread(), recv(), fgets(), etc.

4.5. Conclusions on security

Defining a new low-level language with specific features for memory management is a promising approach to improving the quality of low-level software. So far, research in the area of buffer overruns and integer overflows have focused on either detecting these bugs or adding features to C that specifically deal with these problems. Thus far, these approaches are either incomplete or make software much harder to write. Pondering the larger point of view of using automated memory management, which is a nonspecific tool that's useful in many ways for simplifying code in general, reveals a new point of view for preventing these problems.

Pit avoids infrastructure that usually deters programmers from choosing other languages for low-level code because of high overhead or unpredictable behavior, such as an interpreter, a virtual machine, or a mark-and-sweep garbage collector. There is no performance penalty to using Pit when the programmer chooses to use only primitive variables. When the programmer chooses to use auto variables, there is a cost (how much is yet to be seen). However because of the relation of Zipf's Law to execution time, the penalty of automated memory allocation will not be important in most of the code. When efficiency is necessary, Pit provides an exceptionally easy way to interface auto

variables with optimized code. Perhaps the biggest advantage is Pit's ability to describe time-critical sections in efficient terms while protecting the rest of the code from buffer overruns, integer overflow bugs, non-control-data attacks, double-free bugs, and other memory allocation bugs. These considerations make Pit a superior language for low-level programming.

CHAPTER 5. IMPROVING PORTABILITY

Portability refers to the effort required to transfer a program from one system to another. Another long-standing problem with C is programs usually must be distributed in source form, and the time-consuming process of compiling must be done by end-users. For large applications on a mid-range computer, this can literally take days of CPU time, to say nothing of the time required of people. This is necessary because at compile-time, a program is locked into a particular version of a particular operating system on a particular CPU. There are some systems to mitigate this; for example, the FreeBSD Ports Collection[11] is a repository of software that has already been patched as necessary to allow it to be easily compiled on FreeBSD. This is used to create precompiled packages that may be installed quickly by end-users. NetBSD and OpenBSD, as well as several distributions of Linux, have similar facilities for installing common software. These projects represent an enormous amount of ongoing maintenance, because each port must be individually updated whenever original author releases a new version and because these systems frequently break as a result of trivial differences in the C system library between systems. This often introduces an long delay before users can run new versions of their favorite software.

Unfortunately, the availability of such a porting system has become a major consideration when choosing an operating system. Otherwise superior operating system distributions may be abandoned because they lack strong support for ported software. Many software packages are simply unavailable for many operating system distributions.

Complicating systems further, some operating systems implement awkward compatibility layers for software that is only distributed for other operating systems; for example, FreeBSD implements a compatibility layer for Linux software, but to use this for a typical Linux application, an additional Linux version of every shared library must be installed. Many programs are not available for FreeBSD at all, except by using an executable compiled for Linux on this compatibility layer.

While natively compiled code obviously cannot be portable between hardware architectures, it is not impractical to make object files portable between operating systems on the same architecture. To do so, the system library needs to provide a consistent interface to the operating system, as seen by object files, not just as seen by source code. (Here, interface refers to a set of function prototypes and global variables.) This interface must be extensible in a way that remains back-compatible at the level of machine-code.

The goal is to translate any application into object files just once for each hardware architecture, then distribute the object files to all operating systems for that hardware architecture. The installation procedure links the program and copies it into the appropriate place on the file system. While there will still be portability issues, most notably the locations of files in the file system, this will substantially reduce the effort required when porting software. System-specific configuration issues such as paths can be addressed in other ways. Typically, software installation should entail downloading the software distribution, linking the distributed object files with the system library and third-party libraries, and copying the executable and ancillary files into appropriate locations in the file system.

Traditionally, the number of times a program must be compiled can be described as the product n*m, where n is the number of hardware architectures and m is the number of operating systems. If object files are portable between operating systems, this is reduced to n, because the object files can be compiled once and distributed in compiled form. To do this, parts of the compilation that are dependent on the operating system must be resolved at link-time or be eliminated; resolving them before creating modules locks the modules into a target operating system. While it is unlikely that all portability issues will be resolved for all programs, this approach will significantly reduce the work required for most programs.

As mentioned above, the object files need a consistent interface to the operating system. To accomplish this, the bindings between the application and the system library must be loosened in two important ways at the level of object code:

- The system library cannot define preprocessor macros that are used in the application's modules, because the contents of these macros are almost certain to change. This is generally not a significant issue in Pit because it does not require a preprocessor.
- Variables passed between the system library and applications must be represented in a consistent way. Most importantly, this includes integers and structs.

5.1. Removing the preprocessor

```
1.  #define EMIT_FOO_FLAG(a) ((a) & 0x40 ? puts("True") : puts("False"))
```

Figure 4. A preprocessor macro for a C library that causes dependent programs to lose portability when translated into an object files.

```
1.  int emit_foo_flag(a) {
2.      if(a & 0x40)
3.              return puts("True");
4.      else
5.              return puts("False");
6.  }
```

Figure 5. A C function that does not have the difficulty described in Figure 4.

C's preprocessor carries much blame for binary incompatibility. In C, a library can declare macros that the compiler resolves when the calling module is compiled, and on another operating system, this library may assign different values to these macros. Module interfaces should avoid preprocessor macros in favor of variables declared "constant" and function calls. For example, if the C header file fragment shown in Figure 4 were in a system library, it would cause object files compiled with it to be incompatible with any system that implements the contents of the macro differently (for example, by changing the value 0x40). Changing the preprocessor macro to a function (Figure 5) prevents the compiler from resolving the dependency at compile-time; rather, the dependency is resolved at link-time, and linking can be postponed until after the program is distributed.

Pit does not include a preprocessor. Preprocessor features are obviated in ways that avoid portability problems. There are three actions typically performed by the C preprocessor: substituting constant values, applying in-line calculations, and including header files. The first two are detrimental to portability and are trivial to accomplish with

constant global variables and functions. In place of the third, Pit uses “interface files” to describe the symbols in a module. The compiler automatically generates an interface file when it translates the module the interface file describes. The calling module uses the “import” statement to reference the interface file. The import statement is roughly equivalent to the “#include” statement in C’s preprocessor. (The only substantial difference is that the compiler processes import statements in the same pass as the rest of the code.) While no preprocessor is bundled with Pit’s compiler, it is easy to apply one if desired. As long as the effects of each preprocessor definition are confined to the software package they are defined in, there will be no adverse affect to portability.

5.2. Passing integers portably

```
1.  int fseeko(FILE *stream, off_t offset, int whence);
```

Figure 6. This C function prototype is not portable after translation because the size of off_t will vary between operating systems.

```
1.  public function(in auto $file, in auto $offset, in char $whence) $seek
    prototype;
```

Figure 7. This Pit function prototype is portable because $offset can be any size integer. At run-time, the system library will convert it to what the kernel requires. If the conversion would truncate the value, the system library throws an exception back to the caller. $whence is simply the letter b, h, or e, meaning the offset is relative to the “beginning”, “here”, or “end”. This way, the C preprocessor macros SEEK_SET, SEEK_CUR, and SEEK_END are unnecessary.

In C, when passing variables to functions in the system library, the representation of many integers is system-dependent. For example, off_t and time_t vary between operating systems, and code compiled with one definition of off_t and time_t breaks when moved to a system with a different definition. Solving this requires auto variables. If a parameter to a function is represented as an auto variable, it can be extended without

recompiling existing code because of the structural description auto variables include during runtime. For example, consider file IO functions. Rather than representing the offset into a file as a statically-sized integer as shown in Figure 6, it is represented as an auto variable, as shown in Figure 7. Clearly, this solution will sometimes involve a cost to efficiency. This cost is not as severe as it seems at first, however, because not all values need to be passed in an extensible way. For example, any value that represents something bounded by memory can be represented with an integer whose size matches the hardware's pointer size, referred to as a "naturally-sized integer". Pit guarantees that the size of "int" will always be naturally-sized, so there is no need for the "size_t" and "ssize_t" types provided by C. Thus, any integer representing, for example, an array index, array size, or the size of anything stored in memory can be represented with a naturally-sized integer. If the architecture is extended to a larger pointer size, for example with the recent 64-bit extensions to x86, all the code must be recompiled anyway because the size of a pointer will change. Naturally-sized integers represent most of the integers passed between functions, leaving only a few of them requiring the extensible representation provided by auto variables.

Most numbers that represent the size of an object that cannot fit in memory represent IO routines where the CPU speed is not the limiting factor. Referring back to the file IO example, consider an integer that represents an offset into a file. A file's maximum size is dependent on the operating system, not the CPU architecture. In the not-so-distant past, file sizes were represented with 32-bit variables, limiting files to 4GB. At the time this was decided, it was wasteful to use larger values; an "industrial size" array of hard disks occupying many large machine rooms might approach 4GB. Today, the

industry uses a 64-bit value for representing offsets into files, but it is unreasonable to rely on that being large enough forever, particularly with the proliferation of distributed storage systems such as RAID in the consumer market. In fact, Sun Microsystems has already introduced ZFS, a file system that supports disk arrays larger than 2^{64} bytes in size, and as very few operating systems currently ship with a system library that does not use a 64-bit integer for this, another transition will be necessary eventually. Standard C libraries address this by using off_t to represent file offsets. While this allows the size to be increased without modifying the source code to a program, this is still unsatisfactory, because it requires the compiler to generate different assembler code depending on the target operating system's definition of off_t. Simply compiling two versions is infeasible, because other data types are likely to be in transition as well, each of which doubles the number of compiled versions required. A better solution is to abstract the type of a file offset out of the compiled code entirely (Figure 7). An auto variable can represent the integer without requiring a particular size; the amount of memory allocated by the run-time library will always be sufficient to represent the number.

While representing a file offset with an auto variable reduces efficiently, the only time a file offset is communicated to the system library is when seeking to a new location in the file or querying the size of the whole file. Both of these operations are heavily bounded in speed by their hardware IO requirements (in particular, seeking to a new location on the disk), not the CPU, so the cost should not be measurable. When reading or writing the file, the maximum amount that can be read is bounded by memory, so a naturally-sized integer is guaranteed to be sufficient when representing the amount read or written in a single call. For example, Pit's equivalents to C's read() and write()

functions use a naturally-sized integer since they both require a buffer in memory, but the equivalents to lseek() and fstat() use autos to represent the file size. There may be examples of other system library calls that have not yet been analyzed where the cost to efficiency must be weighed against portability. If optimizing does not yield satisfactory performance for these examples, one possible solution is to provide two interfaces, one that uses auto variables for portability and one that uses fixed-sized primitive integers for speed.

When providing a portable interface, integers may be represented as primitive variables everywhere except where the parameter passes the interface. For example, there is no compelling reason as far as portability is concerned for the system library to use auto variables after receiving a parameter to a function call, because the internals of the system library may easily be changed without changing the API. Also, for situations where the value is otherwise bounded by the design of the caller, the caller may use primitive integers to represent the value. In these cases, the compiler will automatically promote the primitive integer to auto variable when setting up the parameters to the function call and immediately assign the value to a primitive integer in the system library. In this case, the inefficiency is minimized by minimizing the number of operations performed on the variable while it is represented as an auto variable. As long as the parameter passes the library's interface as an auto variable, the interface remains portable.

In short, while there is some cost to using auto variables to pass integers to libraries, the cost can be minimized to a point where it is negligible in most cases.

5.3. Passing structs portably

```
1.  #include <string.h>
2.  #include <sys/types.h>
3.  #include <unistd.h>
4.
5.  /* Network-related includes */
6.  #include <arpa/inet.h>
7.  #include <netinet/in.h>
8.  #include <sys/socket.h>
9.
10. int main() {
11.     struct sockaddr_in my_addr, their_addr;
12.     int sockfd;
13.     char buf[1024];
14.     ssize_t ssz;
15.
16.     /* Prepare my_addr and their_addr, which are temporary structs
    passed to bind()
17.      * and connect().  Notice the very important, but easily forgotten
    call to
18.      * memset().  This is required because implementations may add
    optional fields,
19.      * and we must pass zero for these, not an uninitialized value. */
20.     memset(&my_addr, 0, sizeof(my_addr));
21.     my_addr.sin_family = AF_INET;
22.     my_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
23.
24.     memset(&their_addr, 0, sizeof(their_addr));
25.     their_addr.sin_family = AF_INET;
26.     their_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
27.     their_addr.sin_port = htons(22);
28.
29.     /* Open socket */
30.     if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
31.             perror("socket");
32.             return 1;
33.     }
34.     if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr))) {
35.             perror("bind");
36.             close(sockfd);
37.             return 1;
38.     }
39.     if(connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct
    sockaddr))) {
40.             perror("connect");
41.             close(sockfd);
42.             return 1;
43.     }
44.
45.     memset(buf, 0, sizeof(buf));
46.     if((ssz = read(sockfd, buf, sizeof(buf))) == -1) {
47.             perror("read");
48.             close(sockfd);
49.             return 1;
50.     }
51.     if(write(STDOUT_FILENO, buf, ssz) < ssz) {
52.             perror("write");
53.             close(sockfd);
54.             return 1;
55.     }
56.     if(close(sockfd)) {
57.             perror("remote close");
58.             return 1;
59.     }
60.     if(close(STDOUT_FILENO)) {
61.             perror("stdout close");
62.             return 1;
63.     }
64.
65.     return 0;
66. }
```

Figure 8. A C program for connecting to the local system on port 22 and reading one packet. A well-designed API may improve this, but it would not change the difficulties with portability.

```
1.  import io;
2.  import socket;
3.
4.  public function(out int $exit_status) $.main = {
5.      private auto $s;
6.      $socket.new($s, hash {
7.              "network", "ip4",
8.              "transport", "stream",
9.              "laddress", "127.0.0.1",
10.             "raddress", "127.0.0.1",
11.             "rport", 22,
12.     });
13.
14.     private auto $buf;
15.     $s=>readline($buf);
16.     $io.stdout=>write($buf);
17.
18.     $exit_status = 0;
19. }
```

Figure 9. An equivalent Pit program to Figure 8. In Pit, the error handling may be omitted, because Pit uses exceptions for reporting unhandled errors gracefully. A hash is a robust and simple way to pass all the necessary parameters in one call. Notice that unlike a struct, the order of the elements in the hash does not matter, improving portability. All the parameters passed in the previous C example can be combined into a single hash, greatly simplifying the library interface. Also, either the integer 0x7f000001 or the string “127.0.0.1” may be specified as an IP address, because the library can check whether the auto variable is an integer or string and do the required conversion. As should have been done in C’s libraries, endian conversions for port and network numbers are always done by the library – so there’s no htons(). There’s no need to initialize unused values because a hash keeps track of which values are initialized.

The final difficulty with passing parameters to system library calls in an extensible way is the inflexibility of structs. To be portable, a struct must always have exactly the same format, meaning it must contain the same elements in the same order, each with the same type, with no additional elements. It is a long shot to decide on a format for a struct, and then simply hope that nobody will ever change it and that it will fulfill all future requirements. To do this for all the structs in the system library is clearly impossible. However, the format need not be so rigid, by contrast, if the library uses hashes (as a dynamic type assigned to an auto variable) in place of structs. With hashes, code can check for missing elements, the elements may be in any order, and new code

may add additional elements defined after old code was compiled. For example, if a new version of the system library looks for a new element in a struct, it will simply see whatever uninitialized garbage happens to be in memory past the end of the struct. However with a hash, which elements are present is stored in a dynamic hash table, and the existence of the element is checked at run time. This allows the new system library to fall back to old behavior or assume a default value if a new element is missing. Old code or code compiled on different operating systems remains compatible without recompiling.

Programs written in Perl have had much success with portability, primarily due to the flexibility of Perl's hash data type compared with C's struct data type. Pit's auto variables support the same hash data type as an analog to the struct data type supported by Pit's primitive variables since they are logically quite similar; the only major defining difference is whether the names of the fields can change during run-time. This allows APIs for libraries written in Pit to take advantage of the same benefits to portability that Perl provides.

An excellent example where this is useful is the API for establishing a TCP network connection. This API was extended several times to support new networking protocols, wreaking havoc on portability. In C, the connect() system call requires the programmer to pass a struct containing the family ("AF_INET"), port, and address. Most define additional padding in the struct that reserves space used for operating system specific parameters or other protocols such as IPv6. This space must be zeroed for portability, as shown in Figure 8. All this adds up to an interface that is easier to use wrong than to use right, and is not at all portable after compilation. This is not simply bad

library design. The interface would probably be reasonably easy to use and extend if C provided an extensible representation of data types.

Consider the Pit alternative shown in Figure 9. Here the information is better represented as a hash so the system library can check for unspecified fields (rather than relying on the caller to zero them or reading uninitialized memory), and the system library can take an appropriate default action because all the information about the hash is encapsulated into an extensible self-describing format, even at run-time. In this example, both the local and remote IP address and port are passed in the same call. If the local IP and port are omitted, the system library will simply ask the kernel to automatically select reasonable values. If the remote IP and port are omitted, the system will create a listening socket.

With these portability issues resolved and a reasonably well-designed standard for the system library, object files should be portable between operating systems on the same architecture.

5.4. Conclusions on portability

Improving portability in this way may seem somewhat pedestrian, but C is missing important tools for this job. Adding automatic memory allocation again simplifies code, including some aspects of the system library interface. For example, it obviates the question of whether the calling function or the called function is responsible for freeing memory represented in a return value. (strsep() and scandir() are good examples of the former; getpwnam() and readdir() are good examples of the latter.) For another example, it obviates the possibility of uninitialized elements in structures that appear in parameter lists to system calls. Beyond that, however, the dynamic typing

provided by auto variables abstracts those details of the interface between applications and the system libraries that tend to vary after compilation. This allows us to reduce the difficulty of porting programs to various operating systems significantly.

CHAPTER 6. Z SPECIFICATION

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.

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.

6.1. 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.

6.2. Related work

The formal specification of a programming language[43][44] 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[49][50]. In *[49]*, they used MetaPRL formal programming environment to write a complete compiler for ML-like language. In *[50]*, 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[49][50]. 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.

6.3. 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 *[45][46]*.)

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

6.4. 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 *[46]*, 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

$$\text{allocated\_globs} : \text{GLOBID} \pfun \text{Glob}$$
$$\text{free\_globs} : \text{GLOBID} \pfun \text{Glob}$$

$$\text{dom allocated\_globs} \cap \text{dom free\_globs} = \varnothing \qquad [\text{ No overlap }]$$
$$\text{dom allocated\_globs} \cup \text{dom free\_globs} = \text{GLOBID} \qquad [\text{ All are represented }]$$
$$\forall x : \text{ran allocated\_globs} \bullet x.\text{refcount} > 0$$
$$\forall x : \text{ran free\_globs} \bullet x.\text{refcount} = 0 \land x.\text{value} = \text{undef}$$

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

$$\text{dom allocated\_globs} = \varnothing \qquad [\text{ Implies all are in free\_globs }]$$

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.]
$\forall$ x : dom autos $\bullet$ autos # x = allocated_globs(x).refcount
[This means a Glob that's a reference to a]
[Glob must put that reference in the bag.]
$\forall$ x : { x : ran allocated_globs | x.value $\in$ GLOBID } $\bullet$ x.value $\in$ 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.]

6.5. 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

$$\begin{aligned} &\text{refcount} : \mathbb{N} \\ &\text{value} : \text{Integer} \cup \text{Fraction} \cup \text{String} \cup \text{Array} \cup \text{Hash} \cup \text{Fref} \\ &\hline \\ &\hline \end{aligned}$$

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

$$\begin{aligned} &\text{AllGlobs} \\ &\text{allocated\_values} : \text{VALUEID} \pfun \text{Value} \\ &\text{free\_values} : \text{VALUEID} \pfun \text{Value} \\ &\hline \\ &\text{dom allocated\_values} \cap \text{dom free\_values} = \varnothing \\ &\text{dom allocated\_values} \cup \text{dom free\_values} = \text{VALUEID} \\ &\forall\, x : \text{ran allocated\_values} \bullet x.\text{refcount} > 0 \\ &\forall\, x : \text{ran free\_values} \bullet x.\text{refcount} = 0 \\ &\forall\, v : \text{dom allocated\_values} \bullet \text{allocated\_values}(v).\text{refcount} = \\ &\quad \#\{\, g : \text{dom allocated\_globs} \mid \text{allocated\_globs}(g).\text{value} = v \,\} \\ &\hline \end{aligned}$$

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

value : $\mathbb{Z}$

Fraction

num : $\mathbb{Z}$

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

---

denom > 0

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

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

num > 0 $\Rightarrow \neg \exists$ x : $\mathbb{N}$ • (x > 1) $\wedge$ (num / x $\in \mathbb{N}$) $\wedge$ (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 : ℕ
value : seq ℕ
signed : boolean
-----------------------------------------------
width > 0
if signed then
   ∀ char : value • -2^(width - 1) <= char < 2^(width - 1)
else
   ∀ char : value • 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). ]
∀ x : value • x ∈ 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

$value : String \pfun GLOBID$
$order : seq\ String$

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

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

$dom\ value = ran\ order$ [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

6.6. 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)

6.7. Conclusions on Z specification

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[41]. 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[45]. Along with these advantages, Z notation was effective in representing the auto variable type.

CHAPTER 7. FUTURE WORK

The first work to do next is to write a Pit compiler in Pit. This will both demonstrate that Pit can be self-hosting and improve on the current compiler, which has an overly simple design with no optimizations. It was written for a single purpose, which is to bootstrap the compiler written in Pit.

Then the compiler needs some optimization. Most of the optimization techniques used by C compilers should be useful in Pit. There are additional Pit-specific optimizations. For example, a primitive variable can often be substituted for an auto variable as described above. When work on optimization is complete, it is reasonable to expect Pit programs that use only primitive variables to match the speed of equivalent C programs. Pit programs will always be faster than equivalent Perl programs because when a program written in Perl executes, Perl first compiles it into byte-code, then executes it in a virtual machine. Pit programs will execute significantly faster, even if they only use auto variables, because the code is already compiled at run-time and does not require a virtual machine. This places the performance of Pit programs, at worst, somewhere close to C programs and better than Perl programs.

Next, there needs to be some study of the difference in speed between primitive and auto variables, specifically to determine how high the performance penalty is for using auto variables. This will provide insight into when it’s appropriate to use primitive or auto variables when weighing the trade-off of a program’s performance. Some rough study on this could be done today on toy examples by emulating the logic for auto

variables in C. The examples would be lengthy, inconvenient, and somewhat inconclusive, but with care they could serve the purpose of giving a rough estimate of the difference in speed between the types of variables.

A later useful branch of work may be to create a compiler that translates Pit code into a byte-code that is suitable for a virtual machine. Like Java, this would allow programs to be portable between CPU architectures. It is possible to do this in a way that allows many programs to be compiled natively or for the virtual machine without modification, except programs that use inline assembly code.

Another possible branch of work may be adapting existing research on checking C code for security bugs. While this is obviated for code that exclusively uses auto variables, this research may still be useful to minimize security risks in code that uses primitive variables.

APPENDIX A. GRAMMAR

A.1. Convention

This specification uses BNF, Perl Compatible Regular Expressions (PCRE), and several trivial augmentations to BNF as follows:

- <> surrounds rule names. When empty, represents the null rule.
- "" means literal
- [] means optional (0 or 1 occurrence)
- () means grouping
- means 0 or more occurrences
- + means 1 or more occurrences
- // means comment

A.2. Token separation

The compiler uses PCREs to tokenize the input file. The first capture buffer ($1) captures a token, and the second ($2) captures white-space. If the expression doesn't match, the compiler tries the next expression. If all fail, the compiler reports a syntax error and aborts. If an expression captures only white-space, matching restarts after the white-space with the first regular expression. If an expression captures only a token, it passes the token to the parser, then again restarts matching after the token with the first regular expression. No regular expression matches both a token and white-space. (These

are copied directly from the compiler's source code, with syntactical decoration removed. Assume the s flag to anchor ^ only at the beginning of a string containing newlines.)

```
// Single-quoted strings
^('[^']*')()


// Double-quoted strings may contain \"
^("(?:\\.|[^\\"])*")()


// Single-line comments (treated like white-space)
^()(//[^\n]*)


// Multi-line comments (treated like white-space)
^()(/\*.*?\*/)


// White-space
^()([ \t\r\n]+)


// Bare-words and numbers
^([A-Za-z0-9\.\_]+)()


// Single special character tokens
^([\(\)\[\]\{\}\,\@\#\$\;])()


// Multi special character tokens
^([\'\~\!\%\^\&\*\-\=\+\\\|\:\<\>\/\?]+)()
```

The above identifies boundaries between tokens, but doesn't identify any meaning for the tokens. First, consider individual tokens. Here the specification again uses PCREs for conciseness, but after this section it is restricted to BNF.

```
<number> ::= <integer> | <float>


<integer> ::= ^[0-9]+$
| ^[A-Z0-9]_[A-Z0-9]+$
```

```
<float> ::= ^([1-9A-Z]_)?[0-9A-Z]+
(\.[0-9A-Z]+)?(e[0-9A-Z]+)?$
```

Note that the rules above allow a preceding digit and underscore. This is the notation for expressing a base besides decimal. Any base between 2 and 36 is acceptable. The base is identified by placing the highest digit for that base before the underscore. Here are some examples of <number>:

- A decimal integer: 3735928559
- Another decimal notation: 9_3735928559
- Binary for 3735928559: 1_11011110101011011011111011101111
- Hexadecimal for 3735928559: F_DEADBEEF
- Base-36 for 3735928559: Z_1PS9WXB
- A decimal float: 32.5
- Another decimal notation: 9_32.5
- Binary for 32.5: 1_100000.1
- Hexadecimal equivalent to 32.5: F_20.8
- Base-36 equivalent to 32.5: Z_W.I

```
<quoted-string> ::= ^’[^’]*’$
| ^"(\\.|[^\\"])*"$
```

Single-quotes do not allow special encodings, so it is simpler to use them for strings with literal back-slashes, but it is impossible express single-quotes within a string identified by single-quotes. Double-quotes allow back-slash encodings such as \" \\ \n etc.

Encoding is the only difference between the quoting types. The compiler is 8-bit clean, so a UTF-8 string may appear in quotes with no special support.

```
<bare-word> ::= ^[A-Za-z0-9_.]+$

<symbol> ::= "$" <bare-word>
```

<symbol> is any entry in the symbol table (any variable or function) defined by source code. Note that the leading $ obviates the need for reserved symbol names. Language keywords and numbers such as 32, F_20, if, and asm are distinct from variables and functions named $32, $F_20, $if, and $asm. This allows future versions of the language to include new keywords without breaking old code, which is a problem that C suffers from frequently. While this means there are no reserved symbol names, names should not begin with an underline because these may be used internally by the compiler tools. For example, the symbol "_GLOBAL_OFFSET_TABLE_" is used on many platforms for dynamic linking.

```
<type-def> ::= "@" <bare-word>
```

<type-def> is any name defined by source code to refer to a particular data type. Terminologically, a <symbol> has an associated <type-expression> (which can be a <type-def>), and a <type-def> is a short-hand name for a <type-expression>. Admittedly, both are tracked in the compiler's symbol table, so this specification can be somewhat confusing distinction if unexplained. <symbol>s are variables and functions. <type-def>s

are user-defined types for <symbol>s. As with symbols, names beginning with an underline should be avoided.

```
<jump-label> ::= "#" <bare-word>
```

A jump-label is any name defined by source code to refer to an entry point for execution. These are used for exception handling and for arbitrary jumps. (Pit uses the keyword “jump” instead of “goto” for consistency with assembly language.) The scope of a jump-label is limited to the function it is defined in. These use # instead of $. These labels may be used in expressions to represent a memory address of executable code, which is useful for in-line assembly code. Again, names beginning with an underline should be avoided. The compiler will almost always need to define jump labels for flow control constructs; these are always named with an underline prefix so they don’t conflict with user-defined jump-labels. Unlike GCC, user-defined jump-labels are not renamed because inline assembler code may need to reference them.

A.3. Modules and statements

```
<module> ::= <import-statement>* <statement>*
```

An entire source file is a <module>. <import-statement>s must come before other <statement>s.

```
<import-statement> ::= "import" <bare-word> ";"
```

<import-statement>s simply reference other libraries. Dots in library names form a hierarchical naming convention corresponding to namespaces and subdirectories in the library search path.

```
<statement> ::= <namespace-statement>
| <asm-statement>
| <conditional-statement>
| <loop-statement>
| <exception-statement>
| <declaration-statement>
| <expression> ";"
| "return" ";"


<statement-block> ::= <statement>
| "{" <statement>* "}"


<namespace-statement> ::= "namespace" ":" <statement-block>
| "namespace" <bare-word> ":" <statement-block>
```

A namespace may be anonymous (without a name). Anonymous namespaces have little effect except that the private symbols in them may only be accessed by other symbols within the anonymous namespace. Visibility of public symbols is unaffected by an anonymous namespace. A named namespace has the same effect on private symbols, plus the effect that contained public symbols are all prefixed with the name of the namespace and a dot.

<asm-statement> isn't defined yet. It will represent syntax for specifying in-line assembler code.

```
<conditional-statement> ::= ("if" | "unless") "("
<expression> ")" <statement-block> ["else"
<statement-block>]
<loop-statement> ::= "loop" [<bare-word>] ":"
<statement-block> ["iterate" <statement-block>]
| "again" ";"
| "next" ";"
| "break" ";"
<exception-statement> ::= "try" "#" <bare-word>
<statement-block>
```

```
| "throw" <expression> ";"
| "label" "#" <bare-word> ";"
| "jump" <expression> ";"
<declaration-statement> ::=
<visibility> <type-expression> <type-def> ";"
| <visibility> <type-expression> <symbol> ";"
| <visibility> <type-expression> <symbol> "prototype" ";"
| <visibility> <type-expression> <symbol> "alias" "=" "$"
<bare-word> ";"
| <visibility> <type-expression> <symbol> "=" <initializer>
<visibility> ::= "public"
| "private"
```

<declaration-statement>s define both variables and functions. Functions are declared using initializer syntax, since functions are just constant symbols that contain executable code. Functions are generally a bit of an enigma to syntax, although they are less so in Pit than in most other languages. For example, the sizeof operator does not know how to figure out their size; they are always constant data; the function initializer can only be used in a declaration-statement, not in an generic expression; etc.

A.4. Type-expressions

```
<type-expression> ::= <type-expression-bracket>
| <type-expression-bare>
<type-expression-bracket> ::= "[" <type-expression-bare> "]"
<type-expression-bare> ::= <type-qualifier>*
<type-expression2> <type-expression3>
<type-qualifier> ::= "signed"
| "unsigned"
| "const"
| "unconst"
| "volatile"
| "unvolatile"
| "stateless"
| "unstateless"
<type-expression2> ::= "auto"
| "char"
| "int" ["(" <integer> ")"]
| "ptr" ["(" <type-expression> ")"]
| "float" ["(" <integer> ")"]
| "string" "(" <type-expression> ["," <integer>] ")"
| "struct" "(" (<type-expression> <quoted-string> ",")+ ")"
| "function" "(" (("in" | "out" | "inout") <type-expression>
"$" <bare-word> ",")* ")"
| "@" <bare-word> // user-defined type
| "$" <bare-word> // type of another symbol
<type-expression3> ::= <>
| "->" <type-expression3>
| "<-" <type-expression3>
| "(" ")" <type-expression3>
| "(" <quoted-string> ")" <type-expression3>
| "(" <symbol> ")" <type-expression3>
```

Type-expressions are used for declaring and prototyping symbols, type-casting, within other type-expressions, and the sizeof operator. Note that two of the alternatives in <type-expression3> allow referencing the type of an element in a struct or a parameter to a function. Square brackets are used exclusively to identify a type-expression. For example, a type-cast converting $foo to a 128-bit integer would look like this:

```
$foo[int(128)]
```

In certain cases where the syntax requires a type-expression, the square brackets are required (<type-expression-bracket> is used for these). Specifically, they are required by type-casts and the sizeof operator; they are not required by declaring/prototyping symbols or within other type-expressions.

A.5. Expressions

```
<expression> ::= <value> [<binary-op> <expression>]


<value> ::= <unary-op> <value>
| <value> <postfix-unary-op>
| "sizeof" <type-expression-bracket>
| "(" <expression> ")"
| "(" "if" <expression> "then" <expression> "else"
<expression> ")"
| <symbol>
| "error_id"
| "line_number"
| <number>
| <quoted-string>
| <jump-label>
| ["array" | "hash"] "{" (<immediate> ";")* "}"
<unary-op> ::= "-" | "~" | "not"
<binary-op> ::= "=" | "+=" | "-=" | "*=" | "/=" | "/-="
| "/+=" | "/<=" | "/>=" | "%=" | "|=" | "~|=" | "^=" | "~^="
| "&=" | "~&=" | "<<=" | ">>=" | ":" | "+:" | "-:" | "*:"
| "/:" | "/-: | "/+:" | "/<:" | "/>:" | "%:" | "|:" | "~|:"
| "^:" | "~^:" | "&:" | "~&:" | "<<:" | ">>:" | "or" | "nor"
| "xor" | "nxor" | "and" | "nand" | "<" | "<=" | "==" | "!="
| ">=" | ">" | "<=>" | "min" | "max" | "+" | "-" | "*" | "/"
| "/-" | "/+" | "/<" | "/>" | "%" | "|" | "~|" | "^" | "~^"
| "&" | "~&" | "<<" | ">>"
<postfix-unary-op> ::= "->" // dereference operator
| "<-" // reference operator
| <type-expression-bracket> // typecast
| "(" <expression> ")" // element of a string
| "(" <quoted-string> ")" // element of a for struct
| "(" (<expression> ",")* ")" // function call
```

A.6. Notes

The grammar is mostly presented in an order that minimizes the number of unexplained rules at each point as read from top to bottom. For example, before defining <expression> and referencing many new undefined rules, the grammar ties off all the other loose ends.

One minor but intriguing invention is the syntax for numbers in bases besides decimal. This allows easy representation of any base up to 36, and avoids the confusion caused by the common convention of using a leading zero to designate octal.

At the moment, for-loop syntax is not included. There is some debate over the best syntax for this. Rather than regretting a bad decision later, this matter is unfinished for the time being. Meanwhile, the <loop-statement> syntax allows a clause called “iterate” that mostly covers what for loops can do.

As presented, the grammar appears to support function declarations within functions, however the compiler does not implement this. Also, there are restrictions as to which alternatives for <statement> are allowed inside a function and which are allowed outside a function. This reflects how the proof-of-concept compiler is implemented; code generation, not parsing, enforces these restrictions.

Some details in this grammar still need attention, however there is a functioning compiler that translates the grammar as presented here into i386 assembly language with support for auto variables.

REFERENCES

[1] Dave Ahmad. The rising threat of vulnerabilities due to integer errors. IEEE Security and Privacy, 01(4):77–82, 2003.

[2] Aleph One. Smashing the stack for fun and profit. Phrack http://www.phrack.org/issues.html?issue=49&id=14, 7(49):14, 1996.

[3] J. P. Anderson. Computer security technology planning study. Air Force Electronic Systems Division ESD-TR-73-51, Vols. I and II, 1972.

[4] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In 12th USENIX Security Symposium, August 2003.

[5] Blexim. Basic integer overflows. Phrack http://www.phrack.org/issues.html?issue=60&id=10, 11(60):10, 2002.

[6] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In 14th USENIX Security Symposium, pages 177–192, 2005.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In 7th USENIX Security Conference, pages 63–78, January 1998.

[8] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In 12th USENIX Security Symposium, August 2003.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference & Exposition – Volume 2, pages 119–129, January 2000.

[10] T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The Cornell commission: on Morris and the worm. Communications of the ACM, 32(6):706–709, 1989.

[11] FreeBSD. FreeBSD ports collection. http://www.freebsd.org/ports/.

[12] Peng Li. Safe systems programming languages, October 2004.

[13] OpenBSD. OpenBSD security. http://openbsd.org/security.html.

[14] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. IEEE Transactions on Computers, 55(10):1271–1285, 2006.

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

[16] Leif Pedersen and Hassan Reza. A formal specification of a programming language: Design of Pit. In ISOLA ’06: Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006), pages 111-118, Washington, DC, USA, 2006. IEEE Computer Society.
[17] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. srds, 00:260, 2003.
[18] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In Workshop on Evaluating and Architecting System Dependability (EASY), October 2002.
[19] Michael Zhivich, Tim Leek, and Richard Lippmann. Dynamic buffer overflow detection. In Workshop on the Evaluation of Software Defect Detection Tools 2005, 2005.
[20] Common UNIX Printing System. Project home page. http://cups.org/
[21] Common UNIX Printing System. STR #2919: Multiple vendor CUPS texttops integer overflow vulnerability. http://www.cups.org/str.php?L2919 and http://cups.org/strfiles/2919/str2919.patch
[22] iDefense Labs. Public advisory: 10.09.08. http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=752
[23] Trevor Jim and J. Greg Morrisett and Dan Grossman and Michael W. Hicks and James Cheney and Yanling Wang. Cyclone: A Safe Dialect of C. In ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, pages 275-288, Berkeley, CA, USA, 2002, USENIX Association.
[24] George C. Necula and Scott McPeak and Westley Weimer. CCured: type-safe retrofitting of legacy code. In POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on principles of programming languages, pages 128-139, Portland, Oregon, 2002, ACM.
[25] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Proceedings of RV'03, Boulder, Colorado, USA, July 2003.
[26] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. SIGPLAN Notices, 36(5): 59-69, 2001.
[27] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In In Proceedings of the 11th Annual Network and Distributed System Security Symposium, pages 159-169, 2004. http://citeseer.ist.psu.edu/ruwase04practical.html
[28] Todd M. Austin and Scott E. Breach and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. SIGPLAN Notices, 29(6): 290-301, 1994.
[29] H. Etoh and K. Yoda. ProPolice: Improved stack-smashing attack detect on. IPSJ SIGNotes Computer SECurity 014(025), Oct.2001. http://www.trl.ibm.com/projects/security/ssp
[30] TinyCC. http://tinycc.org/
[31] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. SIGPLAN Notices, 37(5): 13-24, 2002.
[41] Jeannette M. Wing. A specifier’s introduction to formal methods. Computer 23, 9 (1990): 8-22.

[42] NASA. Formal methods demonstration project for space applications — phase I case study: space shuttle orbit DAP jet select. JPL Document D-11432, (1993).
[43] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Technical University of Munich, 1997.
[44] 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): pages 767-775.
[45] J. M. Spivey. The Z notation: a reference manual. Prentice Hall International (UK) Ltd, 1992. (0-13-978529-9).
[46] Jonathan Jacky, The way of Z: practical programming with formal methods. Cambridge University Press, 1996. (0-521-55976-6).
[47] Dick Grune, C. Jacobs, Koen Langendoen, and Henri Bal. Modern compiler design. Wiley, 2001.
[48] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of art and future direction. ACM Computing Survey 28, 4 (1996): 626-643.
[49] 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.
[50] 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).