First-class C Contexts in Cinquecento

Vic Zandy Dan Ridge

IDA Center for Computing Sciences
17100 Science Drive
Bowie, Maryland 20715-4300 USA
{vczandy,dsridge}@super.org

Abstract

Cinquecento is a new functional language designed for debugging and other analysis of software systems comprised of concurrently running C programs. The key innovations of Cinquecento are a first-class abstraction that can represent a C program in execution, a C-based syntactic interface to this abstraction, and language mechanisms for tailoring new instances of the abstraction to unanticipated and heterogeneous environments. Cinquecento embeds these innovations in a conventional Scheme-like functional language. We present the design of the language and demonstrate its usage in the context of program debugging.

1. Introduction

1.1 A Language for Analysis of Heterogeneous Systems

Software systems used in practice often comprise multiple programs, built from different sources, that run and interact concurrently on a network of computers with varying machine and operating system characteristics. Diagnosing anomalous behavior in these systems often involves retrieving and analyzing the run time state of many or all participating programs. *Cinquecento* is a functional language we designed to support such systems analysis.

Cinquecento is based on the observation that the syntax of the C programming language is naturally well-suited for inspecting the run time state of a program in execution. When the target program is itself written in C, retrieving state of interest often involves complex traversals of pointer-based data structures defined by C types. Furthermore, these traversals are often steered by predicates over other program data defined by C types. C is a natural and standard syntax for expressing such logic. Even when the target program is written in some other language, its run time state often consists of binary data structures that can be symbolically modeled by a set of C types and inspected in terms of C pointer operations and iteration constructs. In contrast to previous language-based analysis tools that represent target program state in the special language of the tool [16, 9], Cinquecento provides a representation based on C.

The key innovation of Cinquecento is a new abstraction, called a *domain*, that represents a context in which C expressions can be evaluated. When using Cinquecento to analyze a software system, the user defines a separate instance of the domain abstraction to represent each target program in the system. The user then writes expressions in C syntax to examine the state of target program data structures. Each expression is evaluated by emulating its execution in the context of the target. The resulting value can be stored in Cinquecento data structures for later use by other Cinquecento functions, combined using the ordinary C arithmetic, bitwise, and relational operators with other values computed in the context of the same or different targets, and even modified and copied back to any target.

The domain-based representation of target programs is embedded in a functional language based on the semantics of Scheme. These semantics provide a foundation for structuring complex software analyses. Features important to analysis include high-level data structures such as lists for organizing collected data, mechanisms like closures for sharing arbitrary data structures across disjoint dynamic contexts such as breakpoint handlers, support for incremental and interactive program development, access to host system services, and automatic storage management. A novel aspect of Cinquecento is a language design that integrates statically typed C values and the C forms that operate on them in a dynamically typed functional language.

Cinquecento is designed to cope gracefully with several forms of heterogeneity common in practical software systems. Each target program may define types and symbols that are distinct from (but possibly named the same as) those of the other programs. Each program may be running on a different machine, connected by any of a variety of network services to the others. These machines may vary in their representation of data (*e.g.*, word sizes and byte order), run different operating systems, or have OS version-specific idiosyncrasies in their interfaces for controlling and examining running processes. Previous systems analysis languages impose uniformity assumptions on their targets that make them hard to use or extend in the presence of parts that do not fit the original model. Cinquecento, in contrast, exposes language mechanisms for defining new domain instances tailored for each target.

1.2 Example

To get a feel for the language, consider the task of visiting the nodes of a linked list maintained by a running C program. For the moment, we assume that the program is represented by a Cinquecento domain named `dom`; later we explain how this binding occurs. The following sequence of expressions:

```
mylist = []; /* empty list */
for(p = dom‘listhead; p != NULL; p = p->next)
    append(mylist, p);
```

saves a pointer to each node into the Cinquecento list bound to `mylist`, using essentially the same `for` loop traversal idiom that might be used in the source code of the target. The one difference is

the expression `dom‘listhead`, a reference to the C variable named `listhead` in the program represented by `dom`, whose value we can treat like an ordinary C pointer to the head of the list.

Given a set of target programs represented by a list of domains bound to `mydoms`, we might apply conventional functional programming techniques to repeat the collection operation over the set of programs:

```
mylist = []; /* empty list */
myfn = lambda(dom){
    for(p = dom‘listhead; p; p = p->next)
        append(mylist, p);
};
map(myfn, mydoms); /* list mapping operator */
```

Here each call to `myfn` invoked by `map` binds the `dom` variable to one of the domains in `mydoms`, then performs a C list traversal through the address space of the associated target program. While the traversal idiom is logically always the same, each traversal is tailored by the Cinquecento evaluator to per-target definitions of the `listhead` variable and the type of object to which it points. We assume, for this example, that the programs all define a variable `listhead` that points to an aggregate containing a pointer field named `next`, but all other aspects of these definitions may vary: the address of the `listhead` variable; the offset of the `next` field within the list node; even the name of the type to which `listhead` points. Such flexibility can be useful for validating consistency of a data structure over multiple versions of the same program or over different programs that share data.

1.3 Extending Cinquecento

The domain abstraction hides other details that may vary among the target programs. Each target may be running on a different machine; its domain encapsulates a connection to the right machine. These machines may vary in how they represent C types, such as the size (*e.g.*, 32- or 64-bit) and encoding (*e.g.*, big- or little-endian) of pointers; each domain maintains its own definition of base C type representations. Some machines might run the Windows operating system, while others might run mutually-incompatible variants of Linux or something else; the domain encapsulates the OS-specific protocols and mechanisms for accessing the state of a running process.

The details of many of these variations are too complex, esoteric, and unstable to be baked into an implementation of the Cinquecento language, never mind into its definition. Instead, Cinquecento exposes mechanisms for defining, typically in libraries of Cinquecento code, new custom implementations of the domain abstraction. Users of these mechanisms have full access to all other features of the language, including all data structures, control constructs, and built-in and user-defined functions.

Furthermore, many of the details of writing a custom domain implementation involve manipulation of multiple forms of binary data structures, including executables and other compiler toolchain emissions, data structures exchanged with operating systems, and the contents of binary messages exchanged over networks. These binary objects often encode data structures that are or could be defined as C types (*e.g.*, the ELF format for executables on many Unix variants). Like the state of a running program, these objects often are most naturally manipulated with C syntax.

To support the use of C in the implementation of domains that depend on such binary objects, we recursively allow the domain abstraction to be used to represent them. This treatment follows naturally from our notion of a Cinquecento domain as a representation for *any* context in which C expressions can be evaluated. The memory of a running program, the contents of a binary file, an arbitrary byte string—are each concrete examples of the raw, untyped *address space* that underlies a domain. Abstractly, we view an address space as a procedural interface that serves random access memory read and write operations. Pairing any implementation of this interface with the C types and symbols that define the layout of the data it serves—its *name space*—completes the definition of a Cinquecento domain.

Thus the big idea in Cinquecento is that this interplay of language abstractions for representing C contexts, and the use of C syntax for interacting with these contexts, forms a more powerful substrate for systems programming and analysis than either feature in isolation. Within the same language we can not only *operate* tools like a language-based debugger for software systems, but also *implement and extend* such tools. As a practical matter to fans of functional languages, Cinquecento provides a full set of functional conveniences for doing both jobs.

1.4 Overview

This paper focuses on the language design contributions of Cinquecento: the notion of domains as a representation for C contexts (Section 2), their C-based syntactic and semantic integration in a general purpose functional language (Section 3), and the language mechanisms for defining new domains (Sections 4 and 5).

To illustrate the unusual capabilities of Cinquecento, we present examples of the use of Cinquecento as a debugger for sequential C programs (Section 6).

The design of Cinquecento has been and will continue to be inspired by ideas from previous languages and systems; we recount these influences (Section 7).

Finally, we summarize the contributions and future of Cinquecento (Section 8).

2. What is a Domain?

Conceptually, a domain models an ordinary C execution context as a projection of a set of C type and symbol definitions over a set of addressable storage locations. Every C expression in Cinquecento is evaluated as if it were a fragment of code in a C program modeled by a particular domain. Computed values are the same as those that code emitted by a C compiler would compute under the definitions of pointer size, word size, and byte order specified by the domain, regardless of the characteristics of the machine hosting the Cinquecento evaluation. Operations with storage side effects (*i.e.*, C assignments) cause persistent updates to the domain equivalent to those their execution would cause in a C program modeled by the domain.

Much of the interesting power of Cinquecento stems from the way domains are constructed. The storage associated with a domain, called its *address space*, is defined by a small procedural interface consisting mainly of two random access memory read and write operations. Programs can define new address spaces from arbitrary Cinquecento functions that implement the interface; the language does not care how or where the actual storage operates. In our software analysis application, for example, these functions proxy memory access requests to the process control interface of a remote computer.

Independently, the set of type and symbol definitions of a domain, called its *name space*, is defined by a Cinquecento form whose syntax is based on standard C type and variable declaration syntax. A minor extension to this syntax allows explicit specification of data layout information, such as the exact offset of fields in an aggregate, that C compilers determine implicitly. Additional name space introspection mechanisms allow Cinquecento programs to examine or build new name spaces from the types and symbols of existing ones.

3. Language Tour

Cinquecento is not an experiment in variations on core functional language semantics; rather, it reflects our belief that an established language that has served us well in the past—Scheme—is a good model for a functional language designed for systems analysis. The syntax is radically different and the domain abstraction is new, but the core semantics of Cinquecento should feel familiar to any Scheme programmer.

We summarize the conventional parts. A Cinquecento program consists of a sequence of expressions to be evaluated by the Cinquecento evaluator in order. Variables are dynamically typed, lexically scoped, and can be updated by assignment. Variable bindings can be created in the top-level environment, by a single simple binding form for compound expressions, and by `lambda`. Cinquecento provides several built-in data structures including lists, dictionaries (hash tables), and strings, and a built-in library of supporting functions. It also provides built-in predicates to determine the type of any value, and a library of interfaces to host OS services. And, since no Scheme-like language would be complete without them, Cinquecento provides first-class continuations.

The rest of this section describes how the domain abstraction bridges C syntax and semantics into this functional setting.

3.1 A Simple Domain

We define a simple domain, and its constituent name space and address space, that we will reference throughout this section.

```
ns = @names c32le {       /* name space */
    struct T {
        @0  int id;
        @4  struct T *next;
        @8; /* sizeof(T) */
    };
    @0  int x;
    @4  int y;
    @0  struct T t;
};
as = mkzas(1024);         /* address space */
dom = domain(ns, as);     /* domain */
```

The `@names` form constructs new name space values. Every new name space inherits definitions from a previously defined name space. Here, `c32le` is a built-in name space that defines the base C types as they are commonly defined on 32-bit little-endian machines (*e.g.*, `int` is four bytes). The body of this `@names` expression contains (ignoring the `@` subexpressions) four C declarations, one defining the type `struct T`, and three declaring variables (two `int`s and a `struct T`). The `@` subexpressions exhibit nearly the full extent of our extensions to C declaration syntax to specify layout in an address space (the remaining new syntax is devoted to bitfields).

Address space values are constructed by calling any one of a variety of built-in or user-defined functions. A fresh instance of a zero-filled address space is created by a call to `mkzas`, a built-in function typically used to provide fixed amount (here, 1024 bytes) of backing storage for scratch domains that do not represent external data. Storage in this address space begins at address 0 and is contiguous.

The built-in function `domain` constructs a new domain value from a pair of name space and address space values. Here, `ns` and `as` are Cinquecento variables bound, respectively, to the previously defined name space and address space values, and `dom` is bound to the resulting domain.

Sections 4 and 5 continue the discussion of domain definition. We turn now to the syntax and semantics of C expressions over domains.

3.2 C Values

Most Cinquecento computations begin with a reference to a name in a domain. These are resolved by the backquote operator. For example, the Cinquecento expression:

```
dom‘x
```

yields a reference to the typed location associated with the name `x` in the domain `dom`. This reference is analogous to an *lvalue* in C: a reference through which a stored value may be examined or modified.

The Cinquecento expression

```
a = dom‘x;
```

binds a representation of the referenced value to the Cinquecento variable `a`. Several things happen during the evaluation of this expression:

1. The name `x` is resolved in the domain `dom`, determining the location of `x` (`@0` means byte offset 0) and the name of its type (`int`);
2. Consecutive bytes comprising the value of `x` are read from the address space associated with `dom` starting at the location of `x`. The definition of `int` in the name space of `dom` determines the number of bytes (4) and the encoding of the value (little-endian integer);
3. A *cvalue* is constructed to represent the accessed value. Conceptually a cvalue is a tuple containing the domain, the type name, and the interpretation of the value encoded by the bytes;
4. The variable `a` is bound to the cvalue.

Most C operators that compute or compare values, such as the arithmetic, bitwise, logical, and relational operators, operate over and yield typed values that standard descriptions of C call *rvalues* [8]. Cinquecento cvalues are concrete, first-class representations of C rvalues.

Cinquecento emulates the behavior of all C operators. Operations that yield an rvalue in C yield a cvalue in Cinquecento with equivalent value and type. Overflow and underflow semantics on arithmetic are preserved, the standard promotion and conversion rules are applied to the operands of operators, and so on.

For example, following the evaluation of this sequence of expressions:

```
a = dom‘x;
b = dom‘y;
c = a+b;
```

the variable `c` is bound to a new cvalue that represents the same 32-bit `int` value that would be computed in a C program modeled by `dom`.

As another example, in C the result of a relational operation is an integer rvalue representing the boolean outcome of the comparison. In Cinquecento the result of a relational operator is a cvalue of type `int` with value 0 or 1.

There is an important conceptual difference between C rvalues and Cinquecento cvalues. Rvalues are intangible abstractions for values computed by C expressions. An rvalue can be stored in a typed location in the address space of the program (*i.e.*, by assigning it to a declared variable or a dereferenced pointer), but it cannot be handled directly. Cvalues, in contrast, are first-class values that can be bound to Cinquecento variables. Like other values in dynamically typed languages, cvalues carry their types with them. Similarly, storage for Cinquecento cvalues is like storage for ordinary data in most functional languages: it is unnamed storage implicitly

managed by the Cinquecento implementation. In particular, cvalues do not occupy storage in a domain.

3.3 Pointers

Pointers are values that represent typed locations. Cvalues, in addition to their type, carry a reference to a domain. This reference is used to resolve pointer operations on pointer-typed cvalues to a location in an address space. The expression:

```
p = &dom‘x;
```

binds `p` to a cvalue that represents the location associated with `x` in `dom`. (The precedence of ‘ is higher than that of `&` and most other operators so that domain name references bind tightly.) Specifically, this cvalue comprises:

- a value corresponding to the address of `x` in `dom`;
- the type name `int*`;
- a reference to the domain `dom`.

The C pointer dereference operator `*` is defined in Cinquecento over pointer-typed cvalues. Precisely as a C programmer would expect, `*` accesses the contents of the location in the associated domain, using the underlying type of the pointer, as defined by the domain, to determine the number of bytes to access and their encoding. For example, given the above binding of `p`, either of the two expressions:

```
a = *p;      a = dom‘x;
```

would bind `a` to the same cvalue.

Pointer arithmetic works as expected. The following sequence of expressions:

```
p = &dom‘x;
p = p+1;
```

leaves `p` bound to a pointer cvalue that refers to the location following `x` in `dom`. In C pointer arithmetic, the amount by which the value of a pointer is incremented is determined by the size of the underlying type of the pointer. In Cinquecento, this size is determined by the definition of the type in the domain of the cvalue.

Pointers to aggregate types also work as expected. The following sequence of expressions:

```
p = &dom‘t;
p->next = p+1;
```

assigns the `next` field of the `struct T` at location `t` in `dom` to point to the beginning of the next logical `struct T`. Although there is no symbol corresponding to this location, we can, as in C, use pointers to impute types to otherwise untyped locations.

The C pointer/array relationship works as expected. This sequence of expressions has an equivalent effect to the preceding one:

```
p = &dom‘t;
p[0].next = p+1;
```

The adherence to C pointer semantics extends to disregard for pointer safety. The following sequence of expressions:

```
p = &dom‘x;
p = p+256;
```

binds `p` to a pointer to a location that does not exist in the address space of `dom` (since we used `mkzas` to create a fixed 1024-byte address space).

Such pointers, as in C, are harmless until dereferenced. An attempt to dereference a pointer to an unmapped location in Cinquecento raises a top-level access fault exception that aborts execution of the current Cinquecento program (but leaves the Cinquecento system running). There is no way for a program to catch this or any other Cinquecento exception: they are indications to the user that there is a bug in their Cinquecento program, which they should stop and fix. (Automated recovery from bugs is for robots and spacecraft, not debuggers.) Programs can test, without risk of raising an exception, whether a span of storage in a domain is mapped by calling the built-in function `ismapped`.

Operations on incorrect pointer values in Cinquecento, unlike such operations in C, never cause corruption to the runtime of the Cinquecento system. The environment, control, and other execution state of the Cinquecento system are not represented by a domain, and thus are not addressable data that a Cinquecento program can mutate. (For example, you cannot apply the `&` operator to a Cinquecento variable in hope of discovering its address.) As with other safe functional language runtimes, misbehavior or failure of the Cinquecento runtime is an implementation bug, never a consequence of operating on erroneous user data.

3.4 Assignment

Pointers are commonly used to commit side effects to storage. Following the evaluation of these expressions:

```
p = &dom‘x;
a = dom‘x;
*p = a+a;
```

the `int` stored at the location of `x` in `dom` is replaced with twice its previous value (assuming that the addition does not overflow).

This example illustrates the subtle dual role performed by the C assignment operators in Cinquecento. Compare the above code to these expressions:

```
a = dom‘x;
a = a+a;
```

which perform two updates to the binding of the Cinquecento variable `a`, but do not affect the storage associated with `x` in `dom`.

In general, the effect of the assignment operator depends on the type of its left operand. If the left operand is a Cinquecento variable, then the effect is to update the binding of the variable (it is like `set!` in Scheme). Otherwise, the left operand must be an expression that evaluates to a reference to a domain location, such as a pointer dereference or any other form of a C lvalue, and the effect is to update the bytes at that location.

This rule generalizes to the operands of the other C assignment operators (`++`, `--`, `+=`, `-=`, *etc*). For example, the sequence of expressions

```
a = dom‘x;
a++;
```

leaves `a` bound to the value one greater than the (unchanged) value at `x`, while the following sequence of expressions:

```
p = &dom‘x;
(*p)++;
```

increments the value at `x`, as does

```
dom‘x++;
```

Overloading the assignment operators may seem like a recipe for making unfamiliar code harder to understand. We briefly considered distinguishing the two modes of assignment by creating separate operators for binding updates (like `:=`), but this idea does not map cleanly to the full set of C assignment operators (how would it look for `++`?) and thus would sacrifice useful, established

syntax. Our design is actually less confusing than it might appear: the mode of an assignment operation is always statically determined by the syntax of the left operand; it never depends dynamically on the type of value bound to a Cinquecento variable. In our experience, the intent is always clear and the two modes of assignment quickly become second nature. It would be *more* distracting to have to stop to remember which of two assignment operators to use in different situations.

3.5 Types: Names, Definitions, and Conversions

Cinquecento carefully draws a distinction between the name of a C type and its definition. Type names are syntactic constructions that follow the rules of type specifiers in C declarations, such as `int*`. They determine the set of C operations that may be applied to a cvalue of the named type (such as pointer dereference). The precise byte-level meaning of these operations, however, is determined by type definitions in the name space of the cvalue's domain.

For example, the type name `int*` always refers to the construction of a pointer to a base `int` type, but navigation of this construction depends on a domain name space to provide the size of the pointer and the size and encoding of the `int`. A cvalue of type `int*` in one domain may mean a 64-bit address of a 64-bit integer encoded in big-endian order, while in another domain it may mean a 32-bit address of a 32-bit integer encoded in little-endian order.

A situation in which the distinction matters is printing the type of a cvalue, which can be extracted from a cvalue as a first-class value, called a *ctype*, with the Cinquecento `typeof` operator. The built-in formatted output functions, named `printf` and so on, support a new format verb `%t` that prints ctypes as valid C type specifier syntax. For example, the expression:

```
printf("%t", typeof(&dom‘t));
```

prints `struct T*` to the Cinquecento console. When a type is derived from a `typedef`, this mechanism prints the `typedef` name rather than the type to which the name is bound. The name provides additional semantic information to the user (such as the likely intended use of the value) that would be otherwise lost.

Type conversion occurs in Cinquecento in the same situations as in C: implicitly, such as when the operands of an expression are subjected to C's usual arithmetic conversion rules, or explicitly, when the cast operator is applied to a cvalue. A type conversion changes the name of the cvalue's type, and possibly applies a transformation to the representation of its value. The representation of a value in a cvalue is opaque, so the details of this transformation are up to the implementation. Generally the representation will depend on the definition (not the name) of the type, so the transformation to apply will depend on the definitions of the source and target type names. A type conversion raises an exception if the target type name is not defined in the name space of the cvalue.

The type name to give to the result of an operation is not always obvious. Consider arithmetic of two operands whose type names are two different `typedef`s for `int`. What should the type name of the result be? C offers no guidance: in the context of C evaluation rules, the types of rvalues are intangible, so the type name of this result does not matter. Cinquecento chooses a canonical name for the type definition they have in common (`int` in this case), a more reasonable choice than arbitrarily favoring one of the `typedef` names.

Cinquecento also defines *domain conversion* semantics that map cvalues from one domain to another. One purpose of these semantics is to support the implicit domain conversion rules discussed in the next section. In addition, we have defined an *extended cast* operator, analogous the C's type cast operator, as an explicit language interface to domain conversion. Typically, explicit domain conversion is used to change either the address space of a value or its name space, but not both simultaneously. Section 6 gives examples.

We describe the domain conversion semantics in terms of the explicit domain cast operator. If `v` is bound to a cvalue of domain `dom`, and `dom2` is bound to some other domain, then the expression:

```
v = {dom2}v;
```

performs an explicit domain conversion ({ } is the domain cast operator). The resulting cvalue has domain `dom2`. The type name of the cvalue is unchanged; however, the new *meaning* of the type is the definition of the name in `dom2`. As with ordinary type conversion, the value may be subjected to a representation transformation determined by the new and former definitions of the type.

Another role of the extended cast operator is to perform a C type conversion with a ctype value operand (instead of a literal type name). For example, this sequence of expressions:

```
t = typeof(dom‘x);  /* ctype */
v = {t}v;
```

casts `v` to the type of `dom‘x`.

3.6 Literals and Mixed-domain Expressions

A binary C operator may be applied to cvalue operands from two different domains. Several rules determine the outcome of such *mixed-domain* expressions. The motivation behind these rules is to give sensible meaning to mixed-domain expressions that yield or compare arithmetic quantities, but to disallow rarely sensible mixed-domain combinations of pointer values.

The first rule concerns expressions with literal terms, such as `dom‘x+5`. To maintain a uniform representation for all arithmetic values, Cinquecento represents arithmetic literals as cvalues from a special built-in *literal domain*. This domain is based on a commonly used model of C types on 64-bit machines. Only the size of arithmetic types matters in this domain; pointer size and endianness are irrelevant, because expressions in this domain never reference memory.

When evaluating a mixed-domain expression, the Cinquecento evaluator first checks whether one of the operands is from the literal domain. If so, it implicitly converts the domain of the literal operand to the domain of the other operand. In essence, non-literal values absorb literal values. Undesired truncation of high bits can be avoided by explicitly casting the literal operand to a large type or by employing suffix syntax when expressing the literal (e.g., `0x100000000ULL`).

The choice of 64-bit types for the literal domain matches or subsumes the arithmetic models of any non-literal domain a Cinquecento programmer is likely to create in today's world of 64-bit machines. Should the world move on to C compilers with larger arithmetic models, we can easily enlarge the literal domain to match.

The second rule concerns mixed-domain arithmetic, bitwise, and relational operations on non-literal operands *that are not pointers*. Such operands are both converted to the literal domain. This rule allows comparison or aggregation of arithmetic values from different domains to be expressed naturally without the clutter of explicit domain conversions.

An unintended consequence of the first two rules is that an aggregation of a series of values from different domains can trigger alternating applications of the rules. Consider the following summation over a list of values `vals` from different domains:

```
sum = 0;
map(lambda(val){ sum += val; }, vals);
sum = {@litdom}sum;
```

During the `map` call, the domain of the accumulator `sum` will alternate between the literal domain and that of the most recent non-

literal operand, and the domain of the final result will depend on whether the list had an odd or even number of values. This effect seems harmless to us, since the literal domain subsumes the non-literal domains, however we advise users who care about the resulting domain to explicitly cast the result to the literal domain, which is named `@litdom`.

The third rule concerns pointers arithmetic with mixed-domain operands. Pointer addition operations (including array subscripting) comprise one pointer operand and one arithmetic operand. In mixed-domain pointer addition, the domain of the arithmetic operand is implicitly cast to that of the pointer operand. Since the type of the result is pointer, the domain of the result should remain in the domain of the pointer operand. (This rule also maintains consistency with the rule for literals.) Pointer subtraction operations either involve one pointer operand and one arithmetic operand or two pointer operands. In the former case, the rule is the same as that for pointer addition. The latter case is disallowed: it makes no sense to relate locations in two different address spaces. A program that wishes to do so must explicitly domain cast one of the operands.

The final rule is that all other mixed-domain expressions are disallowed. Generally this rule covers the other mixed-domain pointer expressions. For example, relational comparison between pointers from different domains is prohibited for the same reason that pointer subtraction is prohibited.

3.7 Other Forms

Cinquecento supports C control statements and statement blocks, and Scheme-like closures.

The C control statements, including `if`, `do`, `while`, and `switch`, are defined as *expressions* in Cinquecento. The `continue` and `break` expressions perform the same control transfers within the loop constructs that they do in C. Within any `if` or looping expression, the conditional subexpression that determines control flow must evaluate to a cvalue, which is compared in a domain-insensitive manner to 0 to determine the outcome.

A block of Cinquecento expressions can be enclosed in a compound expression form delimited by { and }. This form introduces a new level of lexical scope; fresh variable bindings may be declared in its body with an optional `@local` form.

The `lambda` form, comprising a formal parameter list and a statement block, yields a first-class function that captures the lexical environment in which the expression is evaluated. Variable-arity functions are supported; the built-in function `apply` applies a function to a list of arguments. The `define` form provides syntactic sugar for creating a function. The `return` form causes control to leave the body of the function, optionally returning a value. We implement Cinquecento function calls with proper tail-recursion [13] to accommodate arbitrary forms of iteration.

3.8 Summary

Before turning to the topic of how domains are defined in Cinquecento, we summarize the major ways in which Cinquecento is *not* like C.

First, as discussed in Sections 3.2 and 3.4, values in Cinquecento are dynamically typed and bound to untyped Cinquecento variables. They do not occupy typed or addressable storage, but rather represent typed scalar values computable in a C domain.

Second, functions are like Scheme closures, not C functions. Cinquecento is designed for examining data in C contexts using C syntax, not for executing previously compiled C code. There is no direct way to define or call a C function in Cinquecento, and Cinquecento functions are not statically typed.

Finally, there is no C preprocessor. The language does not conflict with the standard preprocessor, so experiments in passing Cinquecento code with preprocessor directives through a preprocessor may give satisfactory results; we have not tried. We do miss using the occasional `#define` macro, especially when targeting C programs that make extensive use of such macros. But rather than emulate old macro technology, our future plan is to see whether adopting modern macro technology [3] in Cinquecento can support conventional C macro usage as well as more powerful forms of macros.

4. Name Spaces

A Cinquecento domain is a pair comprising a Cinquecento name space and a Cinquecento address space. We describe the name space abstraction and its definition in this section, and do the same for address spaces in the next section.

A Cinquecento name space models the name space of C variable and type declarations. This includes two disjoint name spaces: the name space for tagged types (`struct`, `union`, and `enum`), and the name space for variable names, base C types, `typedef` names, and `enum` constants.

The key feature of Cinquecento name spaces is that as part of the definition of a name, they include explicit specification of the layout of the defined object in an address space, information that in C is determined implicitly by the compiler. For symbols, the information includes the location of the symbol; for types, it includes the encoding and layout in memory, down to the bit, of the values represented by the type.

Every new name space inherits definitions from a previously defined name space. The built-in *root name spaces* from which all others are derived define base C types for several common machine models. They are bound to the Cinquecento variables `c32le`, `c32be`, `c64le`, `c64be`, `clp64le`, and `clp64be`. Each root name space defines the representation of C base types, including the number of bytes in the type and the encoding of the type in memory. It also defines the size of pointers; for example, `c64le` and `clp64le` define the same commonly used definitions of C types on 64-bit machines, except that the size of pointers is 32-bits in `c64le` and 64-bits ("long pointer") in `clp64le`.

New name spaces are constructed with the `@names` form:

```
@names <expr> {
    <definition> ...
};
```

The `<expr>` is any Cinquecento expression that evaluates to a name space value. Each `<definition>` defines either a type or a symbol. The syntax of these definitions is like the usual C variable and type declaration syntax, but slightly modified to support layout specification. The evaluation of an `@names` expression yields a new name space that contains the definitions in the name space yielded by evaluating `<expr>` plus those specified in the `<definition>`. It is an error to redefine a previously defined name.

There are four forms of name definition: a C aggregate type specification, a symbol corresponding to a C variable, a `typedef`, and an `enum` type specification. Each closely resembles its C counterpart in syntax and semantics. The next four sections describe their use in Cinquecento.

4.1 Aggregate Types

The most interesting type definitions are those for aggregate types (`struct` and `union`). The syntax is like an ordinary C aggregate declaration except that:

- each field declaration includes an offset expression, which specifies the offset of the field relative to the beginning of the aggregate;
- the declaration must end with a final offset expression (not associated with any field) that specifies the size of the aggregate.

The offset expression has two variants: one (`@<expr>`) to specify the offset in *bytes*, and one (`@@<expr>`) to specify the offset in *bits*. The bit offset variant is typically used to define fields corresponding to C bitfields, which can begin at non-byte-aligned bit offsets from the start of the aggregate. For the common case of fields that are not bitfields, bytes are usually the more natural unit for expressing offsets.

Knowledge of the size of an aggregate is used to evaluate pointer arithmetic (and array indexing) operations over aggregates and to define the C `sizeof` of the aggregate. It may seem redundant to explicitly include the total size of the aggregate, but this requirement serves two purposes. First, the size of an aggregate cannot always be inferred from the offset and size of its constituent fields, since compilers may pad the aggregate to satisfy implicit alignment objectives. Requiring the size avoids introducing an explicit notion of alignment. Second, it allows *sparse* definitions of C aggregates in which only fields of interest to the user need be explicitly defined. Fields used for padding or fields corresponding to private (or proprietary) data can be omitted, avoiding the clutter of unused fields.

Cinquecento allows some unusual aggregate layouts that are not expressible in C. For example, a `struct` could have multiple overlapping fields, fields that extend past the aggregate, or (given negative offset values) fields that begin before the aggregate. By default, Cinquecento prints a warning if an `@names` expression specifies layout impossible in C; a user option suppresses the check. However, the ability to define such layout is useful for declaring views of program data structures that are otherwise awkward to specify in the standard C type language. Among other things, it enables an aggregate that is contained in another aggregate to include fields that refer to the containing aggregate or its fields. This can be useful for debugging or validating programs that allocate memory with `malloc`: it allows Cinquecento type definitions for dynamically allocated aggregate objects to include references to the heap meta data that a typical `malloc` implementation prepends or appends to each allocated object.

4.2 Symbol Definitions

A symbol definition assigns a type and a name to a location in an address space, and typically corresponds to a global variable or function in a program executable. The syntax follows standard C variable declaration rules, except that it includes an optional leading offset expression. The assigned type must be the name of a type defined in the name space.

The offset expression specifies the location associated with the symbol; if it is omitted, the definition is still useful for associating a type with a symbol. The form of the offset is any Cinquecento expression that evaluates to an integer cvalue, including non-constant expressions involving Cinquecento variables. (This is true for the offset expressions of aggregates as well.) Usually a name space will define offsets as literal constants, since the locations of symbols are generally statically known (*e.g.*, from the symbol table of an executable) when the name space is defined. Variable expressions are useful, however, in the context of relocatable objects (such as shared libraries) whose symbol tables specify the location of symbols relative to the dynamically selected address at which the library is loaded. For example, this function:

```
define mklibns(loadaddr){
    return @names c32le {
        @loadaddr+0x100 int libfn1();
        @loadaddr+0x200 int libfn2();
    };
}
```

illustrates our approach to defining name spaces for shared libraries. At run time we determine the load address (`loadaddr`) of the library, then pass this address to a function that constructs a name space in which all symbols are statically located in the context of a binding to the value of the load address.

4.3 `Typedef`

`Typedef` is an important type naming operator in C because the names it creates, even when they are simply aliases for base types, cue the reader to the semantic purpose of a variable or field of an aggregate. They can also reduce the clutter of declarations involving multiple tagged types. Cinquecento therefore provides complete support for C `typedef`s.

A well-known issue with `typedef` in parsers for C and C-like languages is that `typedef` dynamically introduces (at parse time) a new token that is grammatically a type name, which then influences how subsequent expressions are parsed. For example, a C expression such as:

```
x = (T)*p;
```

parses differently depending on whether `T` is a variable identifier (to be multiplied, over superfluous parentheses, by `p`) or a type name introduced by a previous `typedef` (to which the result of dereferencing `p` is cast).

In C the answer is statically known by the time the parser reaches the expression. But in Cinquecento the meaning of the corresponding expression:

```
x = (dom‘T)*p;
```

depends on the binding of `T` in the name space of the domain bound to `dom`, which in general cannot be determined until run time.

Our implementation uses a GLR parser to detect and pass to the evaluator ambiguous expressions that depend on the disposition of a name in a domain. Each time an ambiguous expression is evaluated, the Cinquecento evaluator resolves the disposition of its run time dependency (which, although this would reflect poor programming style, need not be the same each time). It then evaluates the corresponding parse of the expression. The overhead of a run time check for these expressions is compensated by the benefit of preserving compatibility with C syntax.

4.4 Enumerations

Like `typedef`, enumeration types provide an important naming mechanism in C programs. Cinquecento provides complete support for C `enum`s. They pose no significant issues in Cinquecento; we discuss them only for completeness.

The syntax for defining `enum` types and their constituent enumeration constants is the same as in C. Enumeration constants belong to the same space of identifier names as symbols and names bound by `typedef`. Like all such identifiers, they can be referenced by an expression of the form `dom‘<name>`, the result of which is a cvalue representing the value of the constant. Following the C standard, the type name of the cvalue is a canonical name for the smallest signed or unsigned base integer type, as defined by the name space, that can represent all constants of the enumeration type.

4.5 Name Space Introspection

Cinquecento defines a built-in library of introspection functions for accessing the set of type and symbol definitions in a name space. The interface can enumerate the set of types and symbols in a name space, and search by name for a type or symbol.

Each type is represented by an instance of the ctype value that was described in Section 3.5. Additional built-in type introspection functions provide access to the components of structured types (including aggregates, pointers, arrays, `enum`s, functions, and `typedef`s), such as the names of fields in aggregates and the subtypes of pointers. There are also type constructors for building new

ctypes from existing ones. These constructors provide programmatic analogues to C type declarations.

Each symbol is represented as a list containing the symbol name as a string, its type as a ctype, and its offset as a cvalue.

There are many uses for the introspection interface. Generally, it is useful for writing Cinquecento programs that automatically explore some aspect of the name space of a target program. For example, we have used it to write functions that print the elements of dynamically linked data structures composed of arbitrary types. Section 6 describes another example.

4.6 Future Improvements

We are considering two improvements to the name space form.

First, the set of root name spaces is currently wired into the implementation. For ordinary debugging applications, this set covers the C type definitions of many contemporary compilers. However, the ability to define new root name spaces would make it possible to accommodate unanticipated name spaces without modifying the implementation. In any event, from a language design perspective, it seems these needlessly second-class data structures should be upgraded to first class. The challenge is to develop a simple and comprehensive Cinquecento syntax for defining root name spaces.

Second, in many cases the layout of an aggregate is predictable. It would be nice in such cases to relieve users (or their name space emitting tools) of the chore of explicitly specifying the layout. Our idea is to allow a user-selected *compiler model* to be associated with a name space definition. The compiler model would encapsulate layout policies used by a particular compiler that the Cinquecento evaluator would query automatically to lay out the aggregates of the name space. We would provide models for commonly used compilers, but we would also allow users to specify their own compiler models. Again, the challenge is to develop a simple and comprehensive syntax for specifying these models.

5. Address Spaces

A Cinquecento address space represents untyped, randomly addressable storage of binary data. There are several built-in constructors for address spaces that represent conventional sources of such data. These include `mkstras`, which converts a Cinquecento string into an address space, `mkfileas`, which maps the contents of a regular file in the local file system as an address space, and the scratch address space `mkzas` (see Section 3.1).

The address space abstraction is defined by a simple procedural interface. Any set of Cinquecento functions implementing this interface can be turned into an address space value. As a result, Cinquecento enables the creation of new user-defined address spaces that represent access to synthetic or remote forms of binary data, possibly defined recursively as other domains.

The interface has three simple operations: one that describes the layout of the address space, and two for reading and writing its contents. The parameters and return values of these operations involve two new Cinquecento types. The first, *address*, represents the address of a storage location; it is just an unsigned integer. The second, *range*, represents a contiguous span of locations; it encapsulates a pair of addresses. Providing special types for such seemingly trivial values enables general composition of address spaces, as we will describe shortly. The storage contents that pass through the interface are an array of bytes represented by the Cinquecento string type.

The operations are:

- $maps : void \rightarrow list\ of\ range$
- $get : range \rightarrow string$
- $put : range \times string \rightarrow void$

An address space need not cover a single contiguous span of addresses, but rather may comprise a sparse collection of disjoint *mappings*. (This is the case for the address spaces of running programs in most operating systems.) The $maps$ operation returns a list of address ranges corresponding to these mappings. This information is typically used to determine whether accessing a particular address would cause a fault. (The `ismapped` built-in function is implemented by invoking the $maps$ operation of the target address space.)

The get operation returns the contents of storage in the requested range. The put operation replaces the contents of storage. If passed an address range that is completely or partially unmapped, both operations call the built-in function `fault` to raise the storage access fault exception instead of returning.

Ordinarily the address space operations are implicitly invoked by the Cinquecento evaluator during its evaluation of C expressions that involve storage access operations (such as dereferencing a pointer). However, the operations may also be directly invoked by name using an overloaded form of the C binary dot (`.`) operator:

```
as.get(mkrange(0,1024));
```

When the left operand of a dot expression is an address space value and the right operand is a name, the expression yields a Cinquecento function representing the named address space operation, which then can be called like any other function. Here, the built-in function `mkrange` constructs the address range argument expected by the get operation; as a convenience `mkrange` promotes its operands to addresses (but they could also be explicitly constructed with the built-in `mkaddr`). As a syntactic convenience, the left operand to the dot operator can also be a domain, in which case the operation is invoked on the underlying address space.

While $maps$, get, and put are mandatory operations (because the Cinquecento evaluator and some built-in functions expect to be able to invoke them), a user-defined address space may be extended with additional special-purpose operations. For example, the address space we define to represent running programs (see Section 6) provides a complete process control interface, including operations for accessing registers, setting breakpoints, and pausing and resuming the process.

The built-in function `mkas` is used to construct a new address space out of Cinquecento functions. Its sole argument is a function of two arguments, conventionally named `dispatch`, that encapsulates the implementations of each operation supported by the address space, including the mandatory ones. The first argument to `dispatch` is the name, as a string, of the operation to be invoked; the second is the list of arguments to pass to the operation. `dispatch` is thus a dynamic dispatch mechanism responsible for forwarding the call to the implementation of the requested operation.

The implementation of the overloaded dot operator distinguishes the name `dispatch` from all others. When passed the name `dispatch`, it simply returns the dispatch function of the address space operand. For all other names, it returns a curried function that calls the dispatch function with the given name as the first argument.

This design supports several general forms of address space extension and composition. First, we can construct a new address space that interposes a filter upon the operations of some existing address space. For example, the following function transforms any address space into one that traces calls made to its operations:

```
define mktraceas(as){
    return mkas(lambda(name, args){
        printf("called %s%a\n", name, args);
        return as.dispatch(name, args);
```

```
        });
    }
```

Second, in such a filter we can additionally override or augment the implementation of a particular operation, or extend an address space with a new operation, simply by trapping calls whose `name` parameter matches that of the target operation. Third, rather than intercept individual operations by name, we can write a filter that traps every operation whose operands or return value are of a certain type, and apply a transformation to those values.

For example, we rely on address space composition to construct a domain that represents the address space of a process that has been serialized into a conventional Unix core dump file. We compose an address space representing the contents of the file with a filter address space that applies a translation function to all address-valued operands, regardless of the operation. The function maps addresses from the address space of the dumped process to offsets in the core dump file. This mapping is determined by a Cinquecento function that interprets, using C types, the section maps of the core dump file.

6. Debugging with Cinquecento

We have implemented a Cinquecento domain that represents the state of a C program in execution on a (possibly remote) Linux machine. It will be a primitive component of a Cinquecento-based software analysis system that we are developing. In the meantime, this domain is also useful for debugging sequential C programs on Linux. We present several examples of the use of Cinquecento in the context of this application.

6.1 Process Domain

The process domain constructor, called `mkprocdom`, starts or attaches to a process on a machine (local or remote) running Linux. For example, the expression:

```
prdom = mkprocdom("node", "/home/user/myprogram");
```

launches a new instance of `myprogram` on the machine named `node`, and binds `prdom` to a Cinquecento domain representing the resulting process. The process is stopped at its first instruction.

To define a name space for the process, `mkprocdom` translates the debugging information embedded in the program executable (*i.e.*, via the `-g` compiler flag) to a Cinquecento name space definition. Currently it performs this translation by piping the contents of the executable through an external C program that reads the debugging symbols and emits Cinquecento code defining the name space. To eliminate the dependence on an external program, we are developing a Cinquecento-based library that can read the debugging sections directly from a domain representing the executable.

The address space of the process domain represents the live memory contents of the process through the standard address space interface. It also extends the interface with several operations specific to process debugging. These include operations for reading and writing registers, setting and clearing breakpoints, and pausing and resuming the process.

The address space is implemented as a set of remote procedure calls over a network connection to a process control server we wrote, called `prctl`. This server runs on the same machine as the target process, and maps the RPCs to calls to the process control interface of the Linux kernel.

It will be easy to implement this domain for other operating systems. Most compilers represent the types and symbols of their executables in a data format that we can parse with a Cinquecento library. The extended address space operations map directly to operations provided by the process control interface of most operating systems. Other than its use of this interface, the `prctl` service has no system-dependent functionality.

6.2 Functional Breakpoints

The typical way breakpoints are used in language-based debugging systems is to schedule user-defined functions to be called when execution of the target program reaches certain code locations. When called, these functions examine and perhaps record some aspect of the state of the process. When they return, the target program resumes execution automatically. We use Cinquecento in this style.

The basic interface is the `bpset` operation of the underlying address space. For example, the following expression:

```
id = prdom.bpset(&prdom‘func, myhandler);
```

sets a breakpoint at the entry of the function named `func` in the process represented by `prdom`. The second argument is a user-defined breakpoint handler function to be called each time the breakpoint is reached. The returned `id` is an integer identifier that can be passed to `bpdel` to clear the breakpoint.

The breakpoint execution model is similar to that of other debugging languages. To start or resume the target process, we call the `cont` operation of the process domain, which directs `prctl` to resume the process and then waits for breakpoint notifications. Each time the target process reaches a breakpoint, `prctl` pauses the process and notifies the waiting process domain. The domain calls the associated breakpoint handler and, unless the breakpoint handler returns the value 0 (an arbitrary choice), directs `prctl` to resume execution of the process. This breakpoint servicing loop repeats indefinitely in the context of the original call to `cont`, until either a handler returns 0 or the process exits.

A common debugging step is to set a breakpoint that is triggered on each call to some function. Often the handler for such a breakpoint accesses the actual parameters passed to the call. Most debuggers implement this access using a primitive built-in mechanism. In contrast, we implement parameter access as a Cinquecento library function called `bpsetargs`.

For example, suppose that a symbol corresponding to the following C function is defined in the process name space:

```
char *func(int arg1, Node *arg2);
```

with `Node` defined as a `typedef` for some aggregate type that includes an `int` field named `id`. Then the following expression:

```
bpsetargs(&prdom‘func,
    lambda(arg1, arg2){
        printf("arg1 = %x, arg2 = %d\n",
            arg1, arg2->id);
});
```

sets a breakpoint at `func` that prints the first argument and the value of the `id` field of the second argument. The handler's arguments, `arg1` and `arg2`, will be cvalues of type `int` and `Node*`, respectively, representing the actual parameters of the current call to `func` in their native types.

The implementation of `bpsetargs` uses name space introspection to determine the number and type of formal parameters expected by `func`. It calls `bpset` to set a breakpoint at the entry to the function with a handler that (1) reads the stack or registers of the process, depending on the function calling convention, to recover the untyped actual parameter values of the call; (2) uses the extended cast operator to promote these values to a list of typed values; and (3) applies the user-supplied handler to this list.

A related debugging step is to set a breakpoint that is triggered on each return from some function. In the handler for such a

breakpoint, it often is useful to be able to relate the value returned to the parameters of the corresponding call. For this purpose we implement another library function, called `bpsetargsret`, that extends the functionality of `bpsetargs` to *optionally* trap, on a per-call basis, returns from calls to the target function. Consider this expression:

```
bpsetargsret(&prdom‘func,
    lambda(retset, arg1, arg2){
        if(arg1 != 0)
           retset(lambda(rval){
                printf("arg1 = %x, arg2 = %d",
                   arg1, arg2->id);
                printf("returned %s\n", rval);
            });
    });
```

The new parameter, `retset`, to the function entry breakpoint handler is a function synthesized by `bpsetargsret` that, when called, sets a *one-shot* breakpoint for the return of the current call to `func`. This breakpoint will remove itself when it fires. It is set only if `retset` is called; as illustrated in this example, logic in the function entry handler may conditionally decide to set this handler based on argument values or other state.

The sole parameter to `retset` is a handler to be called when the function call returns. By re-using the implementation techniques for accessing parameters, `bpsetargsret` arranges for this handler to be passed a cvalue representing the value returned by the function call in its native type. The handler may simultaneously access the parameters of the call through the ordinary variable capture mechanisms provided by `lambda`.

These breakpoint library functions are generic in the sense that they automatically convert the untyped machine-level representation of the actual parameters and return value of the call to correctly typed cvalues. They are currently specialized, however, to the calling conventions of our environment. Any debugger providing similar functionality depends on machine- and compiler-specific knowledge to dissect call frames, trap return points, and recover return values. Unlike most debuggers, we can (and will) generalize our breakpoint functions by linking them to a library of Cinquecento functions that analyze calls for popular target environments, rather than bake these mechanisms into the implementation, where they would be harder for a user to understand, modify, or extend.

6.3 Snapshots

A common operation in systems analysis is taking a snapshot of the state of the system. A necessary (but by no means sufficient) ingredient is a way to snapshot an individual process. Thus we added to the process domain an operation, called `snap`, that returns a new domain that represents the current state of the process frozen in time.

The `snap` operation is also useful for sequential debugging. We give two examples. First, the function call return handler of the previous example captures references to the state of the actual parameters of the call *at the time of the return*, not their state when the function was called. In particular, the dereference `arg2->id` yields the current value of that field. When examining dynamic program state such as a stack trace, the ability to see the original value of arguments passed to functions often yields useful insight into program behavior [10].

Consider the following revision of the previous example:

```
bpsetargsret(&prdom‘func,
    lambda(retset, arg1, arg2){
        if(arg1 != 0){
           snapped = prdom.snap();
           arg2 = {snapped}arg2;
           retset(lambda(rval){
                printf("arg1 = %x, arg2 = %d",
                   arg1, arg2->id);
                printf("returned %s\n", rval);
            });
        }
    });
```

In this version, the return handler will print the value of `arg2->id` *at the time of the function call*. It works by casting `arg2` to a domain representing a snapshot of the process at the time of the call. By simply changing the domain of the pointer, we have redirected the Cinquecento evaluator to access a previous state of the program, without changing the syntax of the dereference operation.

Second, sometimes it is useful to see the value of an argument or some other process state before and after a function call. Suppose, for example, that `Node` is a type that represents a node in a tree data structure whose complex balancing properties must be preserved by any function that mutates the tree, such as an insert operation with the following signature:

```
void tree_insert(Node *root, int val);
```

To validate the effects of this function, we could write the following code:

```
bpsetargsret(&prdom‘tree_insert,
    lambda(retset, root, val){
           snapped = prdom.snap();
           retset(lambda(rval){
                validate({snapped}root, root);
            });
        }
    });
```

where `validate` is a Cinquecento function that checks the balancing invariant over an old and new version of the tree.

Such validation does not require a snapshot mechanism. But without one, we would have to arrange to make a copy of the data structure we were comparing before it is updated. For a large complex data structure, this copy would involve the time overhead of traversing the data structure, the temporary space overhead of storing the copy, and the risk of inaccuracies caused by bugs in our copy logic. In contrast, snapshots provide a simpler, more elegant alternative: a handle to the original.

In low volume, snapshots are inexpensive and reliable to obtain. We implement `snap` by inducing the target process to call the Unix system call `fork`, which creates a copy-on-write clone of the target process. This clone frees all of its external resources (*e.g.*, it closes all of its open files) and it is never allowed to resume execution. When the original process resumes and begins to modify its state, the operating system copies only pages that contain modifications; the others remain shared, even across multiple snapshots. Similar operating system memory management features have been used in previous systems to implement debuggers that can look back in time [6, 12]. However, we believe that this is the first illustration of a language environment for treating these past program states as ordinary C contexts.

In the exclusive presence of one-shot breakpoints, we rely on the garbage collector to keep snapshot volume low. Using a guardian-like mechanism [5], the Cinquecento storage manager detects when there is no longer a live reference to a given snapshot, and automatically directs the remote process control service to destroy the snapshot. The problem of managing larger volumes of snapshot-like resources has already received considerable attention in the large body of literature on debuggers that can go back in time.

6.4 Watching Yourself

An unusual debugging arrangement allowed by our system is a Cinquecento program that monitors its own execution. Such a program can test for unexpected behaviors or regressions in internal mechanisms of the Cinquecento evaluator, such as its garbage collector.

Ideally, a Cinquecento program would be able to set breakpoints in the process in which it executes. This does not work, of course, since arrival at a breakpoint stops the process, which precludes activating a handler. The program would wedge itself at the first breakpoint.

Alternatively, we can set a *snappoint* wherever we would set a breakpoint. Upon arrival at a snappoint, a current snapshot of the process is passed as a new domain to a snappoint handler *in the same process*. In effect, we create a sort of discrete rear-view mirror through which a Cinquecento program can observe precisely rendered past moments of its own execution.

This idea was trivial to implement with the mechanisms described already. We needed to add to the process control protocol served by `prctl` a snappoint request similar to the existing breakpoint request. An actual breakpoint is still placed in the target Cinquecento process. The difference is that when execution arrives at the breakpoint, `prctl` autonomously generates a snapshot, resumes the process, and notifies it of the snappoint event.

This idea is similar to other self-instrumenting systems such as DTrace [2] and kernels based on Kerninst [14], with the advantage that it does not require mechanisms to protect the target system from its own instrumentation.

6.5 Summary

A general theme of the preceding examples is that the Cinquecento domain abstraction, extended with a few primitive debugging operations, enables the advanced capabilities of conventional debuggers—as well as some unusual ones—to be implemented as library code. We could have provided similar examples of Cinquecento code for single stepping, code disassembly, or stack walking. The ability to implement major functionality as Cinquecento code enables users to customize or extend their debugging operations to match their application, and to more easily implement new debugging features.

7. Related Work

Many languages have been developed for analysis of both sequential programs and distributed systems. Cinquecento is distinct for its emphasis on C, rather than a new special-purpose syntax, as the interface to program state, and its supporting language mechanisms for adapting this interface to unanticipated target environments. We compare Cinquecento to two previous systems that were part of the inspiration for Cinquecento, then briefly discuss other closely related work.

7.1 DTrace

The DTrace [2] dynamic instrumentation system, developed originally for system-wide performance analysis for the Solaris operating system, is programmed in a language, called *D*, that provides a C-like language for examining the state of multiple contexts in terms of their native C types and symbols. Cinquecento lifts two ideas directly from D: (1) the use of backquote as an extended C name resolution operator, and (2) the implicit promotion of `char*` values to string values in certain contexts.

The most significant difference between D and Cinquecento is the nature of the systems to which they are intended to be applied. D targets one particular target architecture: the operating system kernel and user programs running on a single machine. D is intimately and implicitly dependent upon the local DTrace system of the target machine for access to the address space, symbols, and types of the target contexts. It lacks a way to bind a D context to a different implementation of these services. Consequently, D cannot be applied to: programs that were not generated by a DTrace-compatible compiler; a remote machine or set of machines; or a machine whose operating system does not support DTrace. In contrast, Cinquecento provides language mechanisms for building C-based analysis interfaces tailored to any of these targets.

A second difference is that D promotes a particular context, that of the OS kernel, as the default frame of reference. Pointer dereference operations, in particular, are always resolved in the kernel address space. To dereference a pointer in a user address space, a D user passes the pointer to a special built-in function that accesses the currently active user address space (the user cannot request a different one). In Cinquecento, no domain is dominant, all domains are named and referenced uniformly, and the *domain associated with each data value* defines the context in which operations on values are performed.

A final difference is the basic language model. D is organized as an awk-like pattern-action language for a rich pre-defined set of execution events, with a set of scoping rules customized for a multi-threaded awk-like execution model. It does not support closures or general lexical scoping like Cinquecento, and does not otherwise resemble a functional language. Both models seem to offer distinct advantages for systems analysis. We find much to admire in the pattern-matching style of D programs, particularly the way its declarative pattern language for specifying events of interest hides the imperative mechanism (*i.e.*, groups of breakpoints) by which these events are trapped. At the same time, we believe that closures and lexical scoping support more easily maintained methods of sharing state across distinct code fragments that respond to related dynamic events (such as the entry and exit of a function call). We plan to experiment with embedding D-like pattern-matching concepts in Cinquecento.

7.2 Acid

Acid [16] is a language-based debugger originally developed for the Plan 9 operating system and subsequently ported to Inferno and many versions of Unix. Cinquecento lifts from Acid the concept of implementing and extending the features of a language-based analysis tool in its own language.

Unlike Cinquecento, Acid is not based on C. Consequently, Acid users express C data structure traversal operations in terms of Acid's types, memory access operations, and looping constructs. While these resemble C, we find that the syntax—especially for referencing variables, dereferencing pointers, and performing iteration—is sufficiently different that Acid often feels like a foreign language.

Acid also has a more limited sense of heterogeneity. While it is possible to apply Acid to a distributed program running on multiple machines—even machines with different architectures—each constituent program must be described by the same name space. Furthermore, Acid depends on the operating system to support a specific interface to remotely running processes; its extensibility does not include new interfaces to program state. Thus Acid cannot be applied to programs distributed over different operating systems.

7.3 Other Work

Many systems have advocated new forms of language support for defining and scheduling responses to events that occur during target program executions [9, 1, 4, 11]. Such features in general provide high-level services for orchestrating complex analyses involving multiple concurrent event streams. While the breakpoint handling mechanisms of Section 6 demonstrate that Cinquecento can support simple forms of event processing, we have not yet adopted

or invented more advanced features on par with these previous systems.

Instead, Cinquecento focuses on a complementary and practical language design problem not addressed by these systems: how to allow users to program responses to events occurring in a heterogeneous set of C programs in terms of symbols, types, and idioms native to each program. Event processing is our next step: in fact, part of the reason that we embedded our solution in a Scheme-based language was to have a standard set of control and data structuring operators with which we and others could model and experience the event processing language constructs proposed in previous work.

Finally, the debugging language DUEL [7] is a unique hybrid: its syntax is based on C, while its semantics are based on a set of Icon-style generators tailored for data structure traversal. The combination provides a concise notation for expressing complex, predicate-driven traversal and enumeration of C data structures. Cinquecento reflects our preference for using standard C idioms to express such logic; the resulting code may be more verbose, but one does not need to learn a new language to understand it. DUEL does not support multiple C contexts nor does it allow to user to implement new representations of target program state.

8. Status

Cinquecento began life as a functional language with a special abstraction designed for analyzing one particular form of data—the state of a program to be debugged. What has emerged, however, is the more general notion of a user-defined context in which C expressions and statements can be evaluated. Cinquecento illustrates a syntactic and semantic design, aggressively faithful to C, that gives this notion concrete first-class status in a conventional functional language setting.

Our design has the direct practical value of allowing a program analysis tool user to use ordinary C syntax to operate simultaneously on multiple distinct programs, and to extend their tool in its own language to operate in new system environments. We also believe that our design may be generally useful in other situations in which structured binary data, of any origin, is manipulated using the types, memory operations, expressions, and control structures of C.

Our ongoing work has two main directions. First, we are completing a prototype of the software analysis system that motivated this work. The main topic of research is incorporating language mechanisms for event processing (as discussed in Section 7.3). Second, we are transitioning our interpreter-based prototype implementation to a better performing incremental compiler. An interesting issue for this compiler is understanding how to generate efficient code for C expressions that are dynamically parameterized by domain-specific definitions of types and storage.

Acknowledgments

Many of our colleagues provided ideas and feedback that significantly influenced and improved the design of Cinquecento and the presentation of this paper. We are grateful for their involvement.

References

[1] M. Auguston, C. Jeffery, and S. Underwood. A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization. In *Proceedings of the Fifth International Workshop on Automated Debugging*, Ghent, Belgium, September 2003.

[2] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June 2004.

[3] R. Culpepper, S. Tobin-Hochstadt, and M. Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, Freiburg, Germany, September 2007.

[4] M. Ducassé. Coca: An Automated Debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[5] R.K. Dybvig, D. Eby, and C. Bruggeman. Guardians in a Generation-based Collector. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI'93)*, Albuquerque, NM, June 1993.

[6] S.I. Feldman and C.B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Madison, WI, May 1988.

[7] M. Golan and D.R. Hanson. DUEL — A Very High-Level Debugging Language. In *Proceedings of the 1993 Winter USENIX Conference*, San Diego, CA, January 1993.

[8] S.P. Harbison III and G.L. Steele. **C: A Reference Manual**, 5th Edition. Prentice Hall, February 2002.

[9] G. Marceau, G.H. Cooper, J.P. Spiro, S. Krishnamurthi, and S.P. Reiss. The Design and Implementation of a Dataflow Language for Scriptable Debugging. *Automated Software Engineering*, **14**, 1, March 2007, pp. 59-86.

[10] R. O'Callahan. Efficient Collection and Storage of Indexed Program Traces. Unpublished manuscript available online, 2007.

[11] R.A. Olsson, R.H. Crawford, W.W. Ho, and C.E. Wee. Sequential Debugging at a High Level of Abstraction. *IEEE Software*, **8**, 3, May-June 1991, pp. 27-36.

[12] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of 2004 USENIX Annual Technical Conference*, Boston, MA, June 2004.

[13] G.L. Steele, Jr. and G.J. Sussman. Lambda: The Ultimate Imperative. MIT Artificial Intelligence Memo AIM-353, March 1976.

[14] A. Tamches and B.P. Miller. Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the Third Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[15] P.R. Wilson and T.G. Moher. Demonic Memory for Process Histories. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI'89)*, Portland, OR, June 1989.

[16] P. Winterbottom. Acid: A Debugger Built From a Language. In *Proceedings of the Winter 1994 USENIX Conference*, San Francisco, CA, January 1994.