Hancock 2.0.1
Manual

July 15, 2002

(Hancock 2.0 : May 15, 2001)
(Hancock 1.1 : March 30, 2001)
(Hancock 1.0 : August 18, 2000)

Kathleen Fisher
Karin Högstedt
Anne Rogers
Frederick Smith

AT&T Labs
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932, USA

hancock@research.att.com

Copyright © 2000, 2001, 2002 AT&T Corp.

Contents

Preface v

Acknowledgements v

New features in Hancock 2.0 v

1 Introduction 1

1.1 What is Hancock? 1

1.2 What Hancock is not 2

1.3 How do I read this manual? 2

1.4 Getting the Hancock compiler 3

1.5 Using the Hancock compiler 3

1.6 Some practical notes about C/Hancock 4

2 "Hello, Hancock!" 5

2.1 Counting international calls 5

2.2 Counting each customer's international calls 8

2.3 Selecting IP packets 12

3 Views 15

4 Multi-unions 19

4.1 Operations 19

5 Hancock data declarations 22

6 Directories 24

6.1 Parameterized directory types 25

6.2 Representations 25

6.3 Operations 26

7 Maps 27

7.1 Type declarations 27

7.1.1 Keys 28

7.1.2 Split 28

7.1.3 Values 28

7.1.4 Defaults 28

7.1.5 Compression 29
7.2 Parameterized type declarations 31
7.3 Operations 33
7.4 Stream conversion 34

8 Pickles **35**
8.1 Type Declarations 35
8.2 Example: Persistent hash table 36
8.3 Type parameters 40
8.4 Operations 41
8.5 Advanced usage 41

9 Streams **42**
9.1 Understanding stream descriptions 42
9.2 Representation 43
9.3 Consuming streams 44
9.4 Event detection 48
9.4.1 Windows 48
9.4.2 Example detection function 48
9.4.3 Example in-line detection code 49
9.5 Writing stream descriptions 49
9.5.1 Runtime model for consuming a stream 51
9.5.2 Declaration forms 51
9.5.3 Example: WorldNet session logs 52
9.5.4 Example: Persistent hash table stream 56

10 Sig_main **58**
10.1 Supported types 59
10.2 Defaults 59
10.3 Boolean flags 60
10.4 Standard input 60
10.5 Program name 61

11 Hancock library **62**

12 Portability issues **65**
12.1 Portable types 65
12.2 Portable format 66
12.3 System-specific definitions 69

13 Extended example: The Cell Tower Application **70**
13.1 Cell Tower: persistent data 71
13.2 Wireless call detail stream 75
13.3 Control flow 75

A Hancock 1.1 versus Hancock 1.0 **84**
A.1 Converting Hancock 1.0 Maps to Hancock 1.1 Maps 85

B Ranges **86**

C ScampRec.hh **87**

D WorldNet.hh **90**

E ipRec.hh **93**

Preface

Acknowledgments

The Hancock project was started in response to a problem posed by Corinna Cortes and Daryl Pregibon about how to write clear but efficient programs to mine transaction streams. We would like to thank them for posing the problem and for helping us understand this domain. Dan Bonachea contributed to the design and implementation of an early version of Hancock. We based the Hancock compiler on CKIT, which is a part of the SML-NJ distribution. Nevin Heintze, Dave MacQueen, and Dino Oliva provided much help with using CKIT. Many people contributed various pieces of software essential to the Hancock system. In particular, David Korn, Glenn Fowler, and Phong Vo wrote the I/O library `Sfio`; John Linderman wrote the sorting library; and Christof Fetzer provided the portability library. Discussions with Dan Suciu and Mary Fernandez helped us refine the stream model. John Reppy produced the pictures used in this manual.

New features in Hancock 2.0

This manual describes Hancock 2.0, the third released version of Hancock. The rest of this section outlines the differences between Hancock 2.0 and Hancock 1.1. Appendix A describes the differences between Hancock 1.1 and Hancock 1.0, the original release of Hancock.

Hancock 2.0 types (directories, maps, pickles, and streams) can be parameterized at runtime. This mechanism reduces the number of distinct types that a programmer must specify without sacrificing the data safety gained from Hancock's static and dynamic type checking. Chapter 5 introduces the notion of parameterized types, while Chapters 6–9 describe the details for each specific type.

Hancock 2.0 maps use a format for specifying keys that separates information about the range of legal keys from the information provided to tune the representation. Previously, this information was encoded in the key specification. Separating it simplifies the specification and reduces errors. Hancock 2.0 maps also allow programmers to specify functions to compress/decompress multiple values at once. Chapter 7 describes maps in detail.

Hancock 2.0 supports Hancock 1.1-style maps, but future versions of Hancock may not. Hancock's range type is supported in Hancock 2.0, but may not be supported in

future versions.

Hancock 2.0 introduces a new type, called generative streams, for streams that do not have a physical representation. It also introduces symbolic names for the return values from stream translation functions and extends the set of possible return values. Streams are described in Chapter 9.

Hancock 2.0 introduces a variety of new features to the `iterate` statement (see Chapter 9). Both the `filteredby` clause and the `withevents` clause can take two forms: the original form that named a function to call and a new expression form. In addition, Hancock now supports a built-in event detection function that generates a single anonymous event for every record in a stream.

Finally, Hancock 2.0 comes with a small library of useful stream types. The library is described in Chapter 11.

Chapter 1

Introduction

1.1 What is Hancock?

Hancock is a C-based domain-specific language designed to make it easy to read, write, and maintain programs that manipulate large amounts of relatively uniform data. Because Hancock is embedded in C, it inherits all the functionality of C. Valid C programs are also valid Hancock programs, and Hancock programs can use libraries written for C. But Hancock is more than C. In addition to C constructs, Hancock provides domain-specific forms to facilitate large-scale data processing.

Hancock's stream construct models data that can be viewed as a sequence of values in a fixed format. Typical examples include records of telephone calls on a long-distance network, session logs from an Internet service provider, and billing records from a credit card company. Hancock constructs make it easy to filter streams to remove unwanted records, to sort streams to improve access locality and hence performance, to detect user-defined events in streams, and to execute user-specified code in response to those events.

Hancock supports three data types—directories, maps, and pickles— for representing persistent data. Directories allow programmers to group related persistent structures, and they provide built-in support for persistence for statically-sized C types. Maps supply a built-in abstraction for associating data with keys. Finally, pickles allow programmers to design their own persistent data types.

For a given data-processing task, Hancock may be suitable if:

1. The task requires a small number of linear passes over a relatively uniform data source.

2. The task requires storing persistent information.

At AT&T Labs we have a suite of Hancock programs that run daily to calculate *signatures* or profiles of AT&T's long-distance customers. These signatures are used for fraud detection and marketing. The programs process roughly nine gigabytes of stream data daily. The most complex application produces a Hancock map that stores values of 120 bytes for over 300 million active keys (from a space of 10 billion possible keys).

These maps require roughly seven gigabytes of space on disk.

1.2 What Hancock is not

Hancock is not a database. Although Hancock maps provide persistent storage, they do not provide many of the services expected of databases, such as support for transactions or declarative query processing. Instead, queries are written as small Hancock programs. The signature suite contains more than a dozen query programs ranging in size from 20 lines to 120 lines of Hancock code.

1.3 How do I read this manual?

In Chapter 2, we illustrate the main features of Hancock by discussing in detail several simple Hancock programs. This chapter serves as an introduction to the kinds of programs that Hancock is designed to support, and it shows how Hancock features can be combined to solve various data-processing tasks. It does not present the various Hancock features in full detail.

Chapters 3 through 10 each focus on a single Hancock construct, explaining that construct using a myriad of examples. Chapter 3 presents the Hancock *view* construct, which specifies two representations for a single piece of data and how to translate between them. In Chapter 4, we introduce *multi-unions*, which are sets of value-carrying labels. Chapter 5 introduces *initializing declarations*, which allow the programmer to connect program variables to data on disk. Chapter 6 presents directories, which are in-memory representations of on-disk directories. We describe maps in Chapter 7. In Chapter 8, we introduce pickles, which provide a way for programmers to specify persistent representations of data structures. Chapter 9 discusses streams. In Chapter 10, we describe the *sig_main* construct, which replaces C's `main` function and provides automatic command-line processing.

The remaining chapters address a variety of topics. Chapter 11 introduces the Hancock library. Chapter 12 discusses issues related to the portability of Hancock programs and data. Finally, Chapter 13 presents an extended example that uses many of Hancock's features.

Where possible, we use examples drawn from an existing application from the telecommunications industry, called Usage, to illustrate the various Hancock features. The Usage program measures the amount of time per day that each telephone number is in active use on a given long-distance network. For each telephone number, the Usage program computes this measure for incoming calls, outgoing calls, and outgoing calls to toll-free numbers.

Appendix A describes the differences between Hancock 1.1 and Hancock 1.0. Appendix B explains Hancock's now deprecated range type. Appendices C, D, and E give Hancock code for describing three data sources. The first source contains *call-detail* data, which describe the telephone calls that occur on a long-distance network. The Usage program uses this data source. The second source contains Internet service provider (ISP) logs, which describe individual connections to an ISP. The third

source describes IP packets. We use these three data sources throughout the manual as examples. We include their definitions in the appendices for reference.

We use the standard C convention of using identifiers with all capital letters for defined constants.

In the remainder of this chapter, we explain how to obtain and use the Hancock compiler, and we discuss some practical issues related to C and Hancock.

1.4 Getting the Hancock compiler

The Hancock compiler is available from the Hancock web page:

```
http://www.research.att.com/projects/hancock
```

Information regarding supported platforms and how to install the compiler may be found there.

1.5 Using the Hancock compiler

This section describes how to use the Hancock compiler `hcc`. The simplest use of `hcc` is to compile a Hancock source file to an executable. The command

```
hcc my.hc
```

will compile the file `my.hc` into C code, compile the resulting C code into object code, and link the object code with Hancock's runtime system to generate an executable called `a.out`. To generate an executable with a different name use the `-o` flag. For example, the command

```
hcc -o my.out my.hc
```

compiles `my.hc` into an executable called `my.out`.

Multiple Hancock source files and C source files may be passed to `hcc`. The compiler will compile each of these files individually based on their extension. Files ending in `.hc` are assumed to contain Hancock code and are compiled as such. Files ending in `.c` are assumed to contain C code and are passed to the C compiler directly. Files ending in other suffixes are not recognized and are simply passed to the C compiler. For object files and libraries, this behavior is usually appropriate. For testing and debugging, it is sometimes desirable to compile the Hancock source file into C code and not generate an executable. The `-C` flag provides this behavior.

To get a brief description of all the flags that `hcc` understands, use the command `hcc --help`. The order of flags is irrelevant. Flags or other command-line options that are not understood by `hcc` are passed through to the C compiler. For instance, to generate an object file instead of an executable, pass the `-c` flag to `hcc`, which will in turn pass it to the C compiler. Unrecognized options are not treated as errors and may result in odd error messages from the C compiler. To obtain a trace of the system commands the compiler executes during compilation, use the `-t` flag. This trace information can help diagnose errors relating to the C compiler and linking.

The following list briefly describes all the available flags:

-s Redirect output to the standard output `stdout`. This option can only be used in conjunction with the `-E` or `-C` flags.

-o Name the output file based on the string following `-o`. This flag can be used to name the output C file or resulting executable depending on what other flags are present. The default output filename depends on the type of the output file. For executables the default is `a.out`. For C files the default is the input filename with its extension replaced by `.c`.

-g Compile all code with debugging information. Link in the debugging version of the Hancock runtime.

-E Only pre-process the input files.

-C Compile the first file with extension `.hc` to a C file and then stop. Compiling to C involves pre-processing the file.

-I Add a directory to search for include files to the call to the pre-processor.

-D Add a definition to the call to the pre-processor.

-m Limit the number of error messages the compiler will generate before giving up.

-t Print a trace of all system commands executed by the compiler.

1.6 Some practical notes about C/Hancock

Hancock allows both by C-style comments (`/* */`) and C++-style comments (`//`). Hancock does not allow the character `$` to appear in identifiers.

The C type `char` defaults to `signed char` or `unsigned char` depending on the implementation. Using `chars` without specifying whether they should be `signed` or `unsigned` can hamper portability.

Hancock does not support the type `long double`.

Hancock uses the `Sfio` library for managing I/O streams. `Sfio` provides functionality similar to that of `Stdio`, the ANSI C Standard I/O library, but via a distinct interface. Sfio provides both source and binary `Stdio` emulation packages. `Sfio` is included in the Hancock distribution. More information about `Sfio` is available from `www.research.att.com/sw/tools/sfio`.

At present there is no debugger designed specifically for Hancock programs, but Hancock programs can be debugged using GDB or DBX with the C code generated by the Hancock compiler.

Please report any bugs in the Hancock implementation or problems with this manual by electronic mail to `hancock@research.att.com`.

Chapter 2

"Hello, Hancock!"

This chapter describes three simple Hancock programs that serve as a gentle introduction to Hancock programming. These programs are analogous to the "Hello, world!" programs familiar from other language manuals— they are designed to familiarize the reader with the basic concepts of Hancock programming. The first of these programs counts the number of international calls seen in a given day on AT&T's long distance network. The second program computes the same information, but on a per phone-number basis. The third prints information about IP packets that were sent to or received from a given set of IP addresses. The purpose of this chapter is to give the reader an idea of what writing a Hancock program entails. We defer to later chapters all the details about the various constructs used in these programs. Chapter 13 contains an extended example that uses many of Hancock's features.

Before discussing these programs, we briefly describe a long-distance call-detail data source. We use this data source frequently in examples, so a passing familiarity with its form will be helpful. Long-distance call-detail contains information about the daily call traffic on a given long-distance network. Each call on the network generates a call-detail record that includes the telephone number of the phone that made the call (the *originating* number), the dialed number, the time at which the call was placed, and the duration of the call.

The example programs presented in this chapter along with a `Makefile` and sample data are available in the Hancock distribution in a directory called `examples`.

2.1 Counting international calls

In this section, we describe a Hancock program that consumes a stream of call-detail records and counts the number of international calls in the process. Figure 2.1 shows the Hancock code that computes this information. To write such an application in Hancock, a programmer needs to:

1. identify a data source and locate the Hancock stream description for it,
2. select the records in the stream that are relevant to the application,

3. identify the interesting events in the stream,
4. describe how to respond to those events,
5. connect program variables with data on disk, and finally,
6. put everything together into a Hancock program.

In the following paragraphs, we walk through these steps for our sample application.

Identify the data source. For our sample application, the data source is a stream of call-detail records. The type `callDetail_s` describes this stream;[1] its definition appears in the header file "`scampRec.hh`" given in Appendix C. For now, we ignore the stream definition itself and focus on the type of the records in the stream:

```
typedef struct {
   pn_t origin;
   pn_t dialed;
   ...
   char isIntl;
} scampRec_t;
```

Each `scampRec_t` describes a call made from an `origin`ating phone number to a `dialed` phone number, each of which has type `pn_t`. In addition to these two fields, the record stores a flag `isIntl` that indicates whether or not the call was international. The call description `scampRec_t` includes other information about the call that we have omitted because it is not relevant to our application. Figure 2.2(a) shows a picture of a stream of call-detail records.

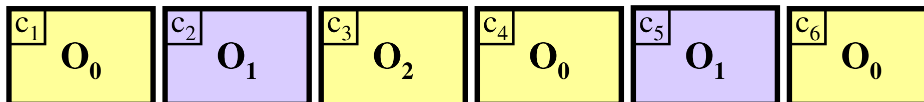
Select the relevant records. We use the `over` clause of Hancock's `iterate` statement (line 9 of Figure 2.1) to indicate that we will be iterating over the call-detail stream `callStream` for our application. Since we want to count international calls, not all the calls in this stream are relevant to our application. We filter out the unwanted calls by specifying a boolean expression in the `filteredby` clause that evaluates to true only for international calls. The syntax `(c)` following the keyword `filteredby` gives the variable name `c` to the current record for use in the filteredby expression. Only those records selected by the filter expression are processed by the remaining code. Figure 2.2(b) shows the sample stream after the domestic calls have been removed.

Identify the events of interest. Hancock's stream processing model is based on detecting and responding to events in the stream. The first step is to specify which events are of interest for a particular application and then to write a function that detects these events given a small window into the stream. For this application, only one kind of event is relevant, namely the occurrence of a call in the stream. This situation is common enough that Hancock provides built-in support for detecting one (anonymous) event per stream record.

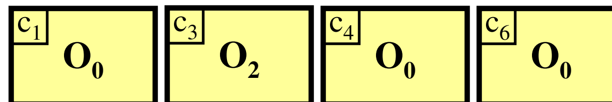
[1]We use the convention that type names ending in a `_s` denote `streams`.

```
#include "scampRec.hh"                                          /* ( 1) */
#include <stdio.h>                                              /* ( 2) */
                                                                /* ( 3) */
void sig_main(callDetail_s callStream <c:>)                     /* ( 4) */
{                                                               /* ( 5) */
  int msgs = 0;                                                 /* ( 6) */
                                                                /* ( 7) */
  iterate                                                       /* ( 8) */
    ( over callStream                                           /* ( 9) */
      filteredby(c) (c->isIntl) ) {                             /* (10) */
                                                                /* (11) */
    event (scampRec_t *c) {                                     /* (12) */
      msgs++;            /* saw an international call   */      /* (13) */
    }                                                           /* (14) */
  };                                                            /* (15) */
                                                                /* (16) */
  printf("There were %d international calls.\n", msgs);        /* (17) */
}                                                               /* (18) */
```

Figure 2.1: Sample application: Counting international calls.



(a) Sample stream of calls.



(b) After filtering.

Figure 2.2: Each box represents a call. The calls are labeled with a call number (c_i) and an originating telephone number (O_i). The dark boxes denote international calls, while the light ones indicate domestic calls.

Events in Hancock carry values. In this case, the carried value will simply be a pointer to the call record (a value of type `scampRec_t`).

Respond to those events. The event clause listed in the body of the iterate statement (lines 12-14) describes how to respond to the anonymous event. The syntax `(scampRec_t *c)` indicates that the variable `c` will contain the value (of type `scampRec_t *`) carried by the event.

To count the number of international calls, this code simply increments the `msgs` counter, which was initialized to `0` just before the `iterate` statement (line 6). Notice that we are guaranteed that any call that triggers the anonymous event will be an international call, because the filter will have removed all other calls prior to event processing.

Connect program variables to on-disk data. The next step is to connect the program variable `callStream` to the corresponding data on disk. Hancock provides the `sig_main` construct as a way to make this connection. The example program takes one command-line argument: a stream of calls (line 4). The program expects the "`-c`" command-line option to name a UNIX file or directory that contains call-detail data. When invoked, the program will start its execution at the beginning of the `sig_main` routine.

Put it all together. Figure 2.1 shows the Hancock program that contains all these pieces. It uses the standard I/O library routine `printf` (line 17) to print the number of international calls that the `iterate` statement detected in the stream `callStream`.

When run in the `examples` directory in the Hancock distribution, the command

```
../bin/hcc -o intlCount.exe intlCount.hc
```

will compile the sample program stored in the file `intlCount.hc` and produce the executable `intlCount.exe`. The command

```
intlCount.exe -c data/calls/binary
```

will invoke the resulting executable on the set of sample calls stored in the file

```
"data/calls/binary"
```

yielding the response:

```
There were 2 international calls.
```

2.2 Counting each customer's international calls

Now that we have written a Hancock program that counts the total number of international calls, we will modify that program to count the total number of international calls made by each "customer," *i.e.*, each telephone number, and store it in a persistent map. We will follow the same sequence of steps as in Section 2.1, with two additions.

First, we will specify the data structures necessary to store the desired information for each customer. Second, we will sort the international call-stream to improve the performance of our program. In the following paragraphs, we discuss this process for our more refined application. The resulting Hancock program is shown in Figure 2.3.

Design a map for storing signatures. The first step is to decide what information to associate with each customer. We call this information the customer's *signature*. Since we want to count the number of international calls made by each customer, the desired signature is an integer. Given this choice, we can define a Hancock *map* type that will associate the desired signature information with each telephone number. The Hancock `map` declaration shown in lines 3-8 of Figure 2.3 defines such a map type, `intlCount_m`.[2] It associates an `int`eger count with each valid phone number. All keys in Hancock are represented using the C type `long long`. The `key` clause in the map declaration indicates that valid keys for this map will fall between `MINVALIDPN` and `MAXVALIDPN-1` inclusive. The `split` clause, explained in detail in Chapter 7, allows the programmer to have some control over the on-disk representation of the data. In this case, the `split` clause indicates that values should be transferred between the processor and the disk in blocks of 10,000 values and that these values should be compressed in units of 1000 values. The `default` clause indicates that each phone number should start with a `default` count of `0`.

Identify a data source, select the relevant records, and sort them. Identifying an input source and deciding how to transform it comes next. Since we will be processing the same kind of stream as in our earlier example, we will again use the `callDetail_s` definition. Figure 2.4(a) contains the sample call stream seen earlier. As before, we will consume this stream using the `iterate` statement (lines 16-35). We will use the same filter expression, `c->isIntl`, because we are interested only in international calls (Figure 2.4(b)). Unlike our earlier example, our more refined application will sort the remaining records by the `origin`ating telephone number using the `sortedby` clause (line 18). Figure 2.4(c) shows the sample stream sorted by originating number. This sorting ensures good locality of access into the Hancock map that stores the customer signatures. Good locality improves program performance, which is a very important consideration given the daily volume of call-detail data.

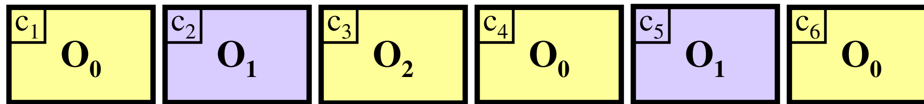
Identify the events of interest. The third step is to identify the relevant events in the transformed stream. Our earlier example had only one event, but this application uses three: whenever a new phone number makes a call (`line_begin`), whenever a call is made (`call`), and whenever a phone number makes its last call (`line_end`).[3] To describe when to raise this event, we give an *event detection* function in the `withevents` clause (line 20). The event detection function is called once for each record. It returns a *multi-union* value that describes the events triggered by that record and gives the values that they carry. The event detection function `originDetect`, which is defined

[2]We use the convention that type names ending with `_m` denote `maps`.

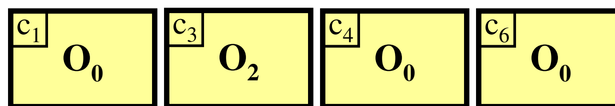
[3]In this context, "first" and "last" are relative only to the immediately surrounding records in the stream, not the entirety of the stream.

```
#include "scampRec.hh"                                   /* ( 1) */
                                                         /* ( 2) */
map intlCount_m {                                        /* ( 3) */
  key (MINVALIDPN .. MAXVALIDPN-1);                      /* ( 4) */
  split (10000, 1000);                                   /* ( 5) */
  value int;                                             /* ( 6) */
  default 0;                                             /* ( 7) */
};                                                       /* ( 8) */
                                                         /* ( 9) */
                                                         /* (10) */
void sig_main(callDetail_s callStream <c:>,              /* (11) */
              intlCount_m ic <I:>)                       /* (12) */
{                                                        /* (13) */
  int msgs;                                              /* (14) */
                                                         /* (15) */
  iterate                                                /* (16) */
    ( over callStream                                    /* (17) */
      sortedby origin                                    /* (18) */
      filteredby(c) (c->isIntl)                          /* (19) */
      withevents originDetect ) {                        /* (20) */
                                                         /* (21) */
    event line_begin(pn_t pn) {                          /* (22) */
      msgs = 0;                                          /* (23) */
    }                                                    /* (24) */
                                                         /* (25) */
    /* saw an international call */                      /* (26) */
    event call(scampRec_t r) {                           /* (27) */
      msgs++;                                            /* (28) */
    }                                                    /* (29) */
                                                         /* (30) */
    /* update the count for pn */                        /* (31) */
    event line_end(pn_t pn) {                            /* (32) */
      ic<:pn:> = msgs + ic<:pn:>;                        /* (33) */
    }                                                    /* (34) */
  };                                                     /* (35) */
}                                                        /* (36) */
```

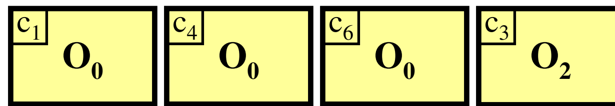
Figure 2.3: Sample application: Counting international calls per customer.



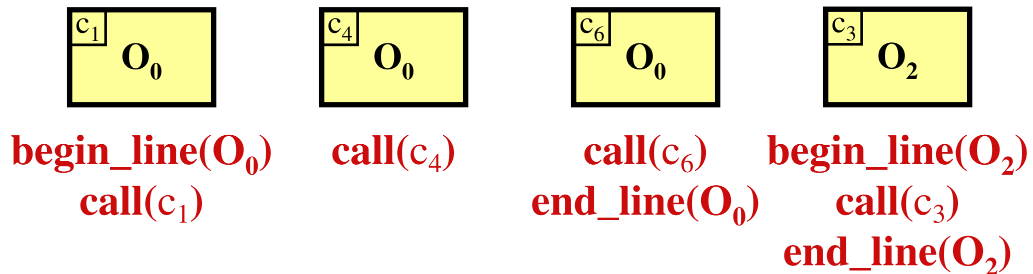
(a) Sample stream of calls.



(b) After filtering.



(c) After filtering and sorting.



(d) After labeling with events using the `originDetect` event detection function.

Figure 2.4: Each box represents a call. The calls are labeled with a call number (c_1) and an originating telephone number (O_0). The dark boxes denote international calls, while the light ones indicate domestic calls.

in Appendix C, detects all three of these events and associates with each such event its "triggering" value. The triggering value for the `line_begin` event is the telephone number that made its first call. For the `call` event, the triggering value is the full record of the call, while for the `line_end` event it is the telephone number that made its last call.

Figure 2.4(d) shows the sample stream labeled with events. The first record will trigger `line_begin` and `call` events because it represents both the first occurrence of a call made by O_0 and a call. The second record will trigger a `call` event. The third will trigger `call` and `line_end` events because it represents both a call and the last occurrence of a call made by O_0. The fourth record triggers all three events because it is the only record for O_2.

Respond to those events. We specify how to respond to these events by listing event clauses in the body of the `iterate` statement (lines 22-24, 27-29, and 32-34). Each event clause names the corresponding event, specifies a type and a name for the value carried by the event, and gives a block of code to execute in response to the event. If the event detection function detects more than one such event for a given record, the event response code is executed in the order that the event clauses appear in the body of the `iterate` statement. For our application, the `line_begin` event clause (lines 22-24) initializes the counter `msgs`. We use `msgs` to count the number of international calls for the "current" phone number, `pn`. The value of parameter `pn` is set to the telephone number that triggered the `line_begin` event. Notice that `msgs` will be reset each time we see a new originating telephone number in the stream. The `call` event code (lines 27-29) increments this counter. The `line_end` event code (lines 32-34) reads the old count for the current phone number from the map `ic` using the notation `ic<:pn:>`, increments it, and then stores the updated count back into the map. For example, consider the three records for calls made by telephone number O_0 described earlier. The first record will trigger `line_begin` and `call` events, which together will initialize `msgs` to `0` and then increment it to `1`. The second record will trigger the `call`, which will increment `msgs` to `2`. The third record will trigger `call` and `line_end` events, which will increment `msgs` to `3`, add this value to the old count for that phone number, and store the result.

Connect program variables to on-disk data. The final step is to connect the Hancock map `ic` to its on-disk representation, which is stored in a UNIX file. As with the stream `callStream`, we use `sig_main` to make this connection. The notation `<I:>` on line 12 indicates that the name of the file to hold the resulting `intlCount_m` will be supplied using the "`-I`" command-line switch when the program is invoked.

2.3 Selecting IP packets

In this section, we describe a Hancock program that consumes a stream of IP packets, printing information about the packets that were either sent to or received from a given set of IP addresses. Figure 2.5 shows the Hancock program for performing this oper-

```
#include "ipRec.hh"                                           /* ( 1) */
#include "ihash.h"                                            /* ( 2) */
                                                              /* ( 3) */
hash_table *ofInterest;                                       /* ( 4) */
                                                              /* ( 5) */
int inSet(ipPacket_t *p)                                      /* ( 6) */
{                                                             /* ( 7) */
   if (hash_get(ofInterest, p->source.hash_value) == 1)       /* ( 8) */
      return 1;                                               /* ( 9) */
                                                              /* (10) */
   if (hash_get(ofInterest, p->dest.hash_value) == 1)         /* (11) */
      return 1;                                               /* (12) */
   return 0;                                                  /* (13) */
}                                                             /* (14) */
                                                              /* (15) */
void sig_main(ipAddr_s addrs <l:>,                            /* (16) */
              ipPacket_s packets <p:>)                        /* (17) */
{                                                             /* (18) */
  /* code to set up hash table */                             /* (19) */
  ofInterest = hash_empty();                                  /* (20) */
                                                              /* (21) */
  iterate                                                     /* (22) */
    ( over addrs ) {                                          /* (23) */
                                                              /* (24) */
    event (ipAddr_t *addr) {                                  /* (25) */
      if (hash_insert(ofInterest, addr->hash_value, 1) < 0)   /* (26) */
         exit(1);                                             /* (27) */
    }                                                         /* (28) */
  }                                                           /* (29) */
                                                              /* (30) */
  /* code to select packets */                                /* (31) */
  iterate                                                     /* (32) */
    ( over packets                                            /* (33) */
      filteredby inSet ) {                                    /* (34) */
                                                              /* (35) */
    event (ipPacket_t *p) {                                   /* (36) */
      printPacketInfo(p);                                     /* (37) */
    }                                                         /* (38) */
  };                                                          /* (39) */
}                                                             /* (40) */
```

Figure 2.5: Sample application: selecting IP packets.

ation. Our implementation of this application uses two streams. The first represents a collection of IP packets using the following Hancock definitions:

```
typedef struct{
  unsigned int ts1, ts2;
  unsigned char version;
  ...
  ipAddr_t source;
  ipAddr_t dest;
} ipPacket_t;
stream ipPacket_s { getValidIPPacket : Sfio_t => ipPacket_t;};
```

Each ipPacket_t packet contains a time stamp, the version of IP used by the packet, source and destination IP addresses, as well as other information. For our purposes, only the source and destination of the packet are relevant. We represent IP addresses using four unsigned char's to store the four pieces of an IP address and an additional field hash_value that merges these four values into a single long for hashing purposes. The hash values may also be used as map keys for other applications.

The second stream represents the collection of IP addresses about which we want to collect information. The following Hancock declaration defines this stream:

```
stream ipAddr_s { getvalidIPaddr : Sfio_t => ipAddr_t;};
```

Each element in this stream represents a single IP address. The complete definitions for both streams appear in the header file "ipRec.hh", given in Appendix E.

Our implementation of this application has two steps. First, it builds a hash table to store the IP addresses of interest. Second, it checks the source and destination of each packet in the packet stream. If either address appears in the hash table, it prints out the desired information about the packet.

The code in lines 22-29 of Figure 2.5 builds the hash table by iterating over the addrs stream. It uses the built-in anonymous event-detection function to generate one event for each IP address in the stream. For each such event, the response code inserts the IP address into the hash table. The function hash_insert will fail by returning a negative number only if the table runs out of space, in which case the program exists.

Once the table is built, the implementation walks over the stream of packets using the iterate statement in lines 32-39. This statement uses a filter function, inSet (lines 6-14), rather than a filter expression, to keep only packets that have a source or destination that is of interest. The function works by looking up the source and destination of the packet in the hash table. Only packets that have at least one successful lookup pass the filter. The second iterate statement also uses the built-in anonymous event detection function to generate an event for each packet in the filtered stream. The event response code prints the desired information about the packets. Note that only packets that have a source or destination address that is of interest will reach the event response code because the filter function eliminates all other packets.

As in our previous examples, the final step is to connect program variables with data on disk using Hancock's sig_main construct. The code in lines 16-17 indicates that a user will specify the stream of addresses of interest (addrs) using the -l switch and will specify the packets (packets) using the -p switch.

Chapter 3

Views

Hancock provides the `view` construct to allow programmers to specify two views of a single piece of data and how to convert between those views. The `view` declaration introduces three type names, one for the `view` type and one for each of the constituent views, which we refer to as *sub-views* when our usage might not be clear from the context.[1] The `view` construct defines the types for the sub-views, and it specifies *conversions* for switching between them.

The following simple `view` declaration specifies two views of a unit of time: one represents time as seconds, the other represents time as bucket numbers. Each bucket corresponds to a range of seconds.

```
view time_v(sec_t, bucket_t) {
  double <=> char;
  sec_t(b) { return bucketToSec[b]; }
  bucket_t(s) { return secToBucket(s); }
}
```

This declaration introduces three type names:

- `time_v`: a name for the view type,
- `sec_t`: a name for the left-hand sub-view (`double`), and
- `bucket_t`: a name for the right-hand sub-view (`char`).

The view type, `time_v`, is optional and can be used only in other view definitions. The sub-view types, `sec_t` and `bucket_t`, can be used like regular C types in variable declarations, as parameters, *etc.* The underlying types for the sub-views are specified on either side of the `<=>` operator. In this example, we represent seconds using a `double` and bucket numbers using a `char`.

The `sec_t(b) {...}` portion of the view declaration specifies how to convert `bucket_t b` to produce a `sec_t` value. Similarly, `bucket_t(s) {...}` specifies

[1] We use the convention that type names ending in a `_v` denote `views` and that type names ending in a `_t` denote structure types, include the two sub-views of a `view` type.

how to convert a `sec_t s` to obtain a `bucket_t`. In this example, `sec_t(b)` uses the array `bucketToSec` to convert a bucket `b` into the corresponding mean number of seconds for that bucket. Conversion `bucket_t(s)` uses the function `secToBucket` to convert the seconds stored in double `s` into a bucket number. Information will be lost converting between these two views because a single bucket represents a range of times rather than a single time. Using views to specify the correspondence between an exact value and an approximation of that value is common in Hancock programs.

Hancock provides the view operator ($) to convert between views. For example, the following code converts from bucket to a number of seconds and then back using the view operator:

```
bucket_t b = 3;
sec_t s;

s = b$sec_t;
b = s$bucket_t;
```

Views often relate exact and approximate data. As a result, the conversions may not be idempotent. For example, depending on how the conversions for the view `time_t` were written, the value of `b` at the end of the computation may not be three.

Views can be more complex than the above example would indicate. In particular, the view `time_v` has only one *field*. In the general case, views can have more than one field, in which case the fields are named. For example, the following view contains fields for three second/bucket values.

```
view (uSig_t, uApprox_t) {
  double <=> char in;
  double <=> char out;
  double <=> char outTF;

  uSig_t(ua) {
    uSig_t us;
    us.in    = bucketToSec[ua.in];
    us.out   = bucketToSec[ua.out];
    us.outTF = bucketToSec[ua.outTF];
    return us;
  }

  uApprox_t(us) {
    uApprox_t ua;
    ua.in    = secToBucket(us.in);
    ua.out   = secToBucket(us.out);
    ua.outTF = secToBucket(us.outTF);
    return ua;
  }
}
```

As illustrated by this declaration, the name of a `view` type can be omitted. The sub-view types `uSig_t` and `uApprox_t` are equivalent to C structures constructed from the left and right sides of the <=> operator:

```
typedef struct {
   double in;
   double out;
   double outTF;
} uSig_t;

typedef struct {
   char in;
   char out;
   char outTF;
} uApprox_t;
```

These types are available to the view's conversions. The types uSig_t and uApprox_t are used in the Usage program. uSig_t represents an "exact" view in seconds of incoming, outgoing, and tollfree usage, while uApprox_t represents an approximate view of the same data.

In addition to having fields defined with the <=> operator, view fields can be defined using standard C types or even other views. Using views to define fields is convenient when multiple fields have the same structure. For example, the above view could be declared using the time_v view with the following alternative syntax:

```
view (uSig_t, uApprox_t) {
  time_v in;
  time_v out;
  time_v outTF;

  uSig_t(ua) {
    uSig_t us;
    us.in    = bucketToSec[ua.in];
    us.out   = bucketToSec[ua.out];
    us.outTF = bucketToSec[ua.outTF];
    return us;
  }

  uApprox_t(us) {
    uApprox_t ua;
    ua.in    = secToBucket(us.in);
    ua.out   = secToBucket(us.out);
    ua.outTF = secToBucket(us.outTF);
    return ua;
  }
}
```

If the programmer does not specify conversions, Hancock will construct them automatically on a field-by-field basis. If a field f has a view type v, Hancock will use the conversion functions associated with view v. If f has a standard C type, Hancock will use the identity function for that field. Using this facility, the above example could be rewritten:

```
view (uSig_t, uApprox_t) {
  time_v in;
  time_v out;
  time_v outTF;
}
```

Given this declaration, the Hancock compiler would construct the following conversions automatically:

```
uSig_t(a) {
  uSig_t s;
  s.in    = bucketToSec[a.in];
  s.out   = bucketToSec[a.out];
  s.outTF = bucketToSec[a.outTF];
  return s;
}

uApprox_t(s) {
  uApprox_t a;
  a.in    = secToBucket(s.in);
  a.out   = secToBucket(s.out);
  a.outTF = secToBucket(s.outTF);
  return a;
}
```

Chapter 4

Multi-unions

Hancock supports a type called a *multi-union* that can be thought of as a finite set of tagged values. The set of legal tags (or *labels*) and the types of the values associated with the tags are fixed in the type definition, but a value of the type may contain any subset of the tags. For example, the multi-union declaration:

```
munion line_e {: areacode_t npa_begin,
                 exchange_t nxx_begin,
                 pn_t line_begin,
                 scampRec_t call,
                 pn_t line_end,
                 exchange_t nxx_end,
                 areacode_t npa_end :};
```

creates an multi-union type line_e.[1]

Expressions of type line_e can be constructed directly. For example, if ac has type areacode_t[2]and lp has type pn_t, then the expression

```
{: npa_begin = ac, line_begin = lp, line_end = lp :}
```

creates a value of type line_e. The empty set, {: :}, is a legal value of any multi-union type, but it must be cast explicitly to have the appropriate type (for example, (line_e) {: :}).

4.1 Operations

The multi-union type supports a variety of operations: membership test, value access, union, difference, and removal. Figure 4.1 contains code that illustrates these various operations.

The membership operation, written @, takes a multi-union and a label as operands. It returns true if the label appears in the multi-union value, and false otherwise. For

[1]We use the convention that type names ending in an _e denote multi-unions.

[2]The types areacode_t, exchange_t, pn_t, and scampRec_t are defined in Appendix C.

```
pn_t pnL, pnR;                                                          /* ( 1) */
exchange_t xL, xR, x1, x2;                                              /* ( 2) */
areacode_t ac;                                                          /* ( 3) */
line_e l, r, u, d;                                                      /* ( 4) */
char bL, bR;                                                            /* ( 5) */
                                                                        /* ( 6) */
l = {: nxx_begin = xL, line_begin = pnL :};                             /* ( 7) */
r = {: line_begin = pnR, nxx_end = xR :};                               /* ( 8) */
                                                                        /* ( 9) */
bL = l@nxx_end;       /* false */                                       /* (10) */
bR = r@nxx_end;       /* true  */                                       /* (11) */
                                                                        /* (12) */
x1 = l.nxx_end;       /* undefined */                                   /* (13) */
x2 = r.nxx_end;       /* xR        */                                   /* (14) */
r.npa_begin = ac;     /* {: npa_begin = ac, line_begin = pnR, */        /* (15) */
                      /*    nxx_end = xR :}                   */        /* (16) */
                                                                        /* (17) */
u = l :+: r;          /* {: npa_begin = ac, nxx_begin = xL,    */       /* (18) */
                      /*    line_begin = pnR, nxx_end = xR :} */        /* (19) */
d = l :-: r;          /* {: nxx_begin = xL :}   */                      /* (20) */
                                                                        /* (21) */
l :\: nxx_begin;      /* {: line_begin = pnL :} */                      /* (22) */
```

Figure 4.1: Multi-union operations.

example, the value of `bL` will be false and the value of `bR` will be true after lines 10 and 11.

Hancock uses "`.`" as its value access operation for multi-unions. This operator takes a multi-union and a label as operands. It can appear as either an r-value or an l-value. Used as an r-value, the access operation returns the value associated with the label if the label appears in the multi-union operand and an undefined value otherwise. For example, the value of `x1` will be undefined and the value of `x2` will be `xR` after lines 13 and 14. Used as an l-value, the access operation allows the programmer to change the value associated with a label in a multi-union or to add a new label to a multi-union. For example, the assignment in line 15 adds the label `npa_begin` with the value `ac` to the multi-union `r`, resulting in a value of:

```
{: npa_begin = ac, line_begin = pnR, line_end = xR :}
```

The multi-union union operation, written `:+:`, takes two operands that have the same multi-union type and produces a new value that contains the union of the labels of the two operands. A label carries its value into the union unless it appears in both operands, in which case it takes its value from the right-hand operand of the union. For example, the union operation in line 18 will yield a value of:

```
{: npa_begin = ac, nxx_begin = xL,
   line_begin = pnR, nxx_end = xR :}
```

for the variable `u`. The label `nxx_begin` and its value come from the variable `l`.

The remaining labels and their values come from `r`, including `line_begin`, which appears in both `l` and `r`, because the union operation is right dominant.

The multi-union difference operation, written `:-:`, takes two operands that have the same multi-union type and produces a new value of that type. The new value contains the labeled values from the first operand that do not appear in the second. For example, the difference operation in line 20 will result in variable `d` getting the value `{: nxx_begin = xL :}`.

The multi-union remove operation, written `:\:`, takes a left operand of multi-union type and a right operand that is a label. If the label appears in the multi-union, the removal operation destructively removes it and its associated value. The code in line 22 in Figure 4.1 removes the label `nxx_begin` and its associated value from `l`, which yields a value of `{: line_begin = pnL :}`.

Chapter 5

Hancock data declarations

Hancock supports four mechanisms for managing data: directories, maps, pickles, and streams. These mechanisms differ in their details, but they share a common mechanism for declaring program variables and for connecting those variables to persistent data on disk. This chapter explains how to declare such variables. We leave the details of declaring these types to the next four chapters.

Once the programmer has specified a Hancock data type (dir_d, for example), uninitialized variables of that type can be declared using the usual C syntax (dir_d d). This mechanism alone is not sufficient, however, because the programmer must be able to connect program variables to data on-disk. Hancock provides two mechanisms for making this connection: initializing declarations and sig_main parameters. We describe the first of these mechanisms here and defer the explanation of the second to Chapter 10.

An initializing declaration augments a standard C declaration with additional information, in particular, with qualifiers, actual parameters, and a location. The following declaration, for example:

```
const exists cellTower_d cellData(:9:) = "cellData.current";
```

declares that variable cellData has type cellTower_d with an actual type parameter of 9. It also declares that the variable cellData should be connected to the data stored on disk in "cellData.current", which will not be updated (const) and must exist on disk (exists).

To promote data safety, Hancock provides three qualifiers for type declarations: const, exists, and new. These qualifiers convey the programmer's expectations about how a data structure will be used. The runtime system generates an error and halts the program when a property specified by a qualifier is violated at runtime. The const qualifier indicates that the data structure will not be updated during the computation. Unlike the const qualifier of plain C, the const qualifier in Hancock's initializing declarations cannot be cast away. The exists qualifier indicates that the specified on-disk representation must exist on disk at runtime. The new qualifier has the opposite effect; it indicates that the specified on-disk representation must not ex-

ist at runtime.[1] If the programmer does not specify either `exists` or `new`, then the runtime system assumes that the programmer does not care whether the on-disk representation exists. In this case, the runtime system uses the existing representation if one exists or creates a new one otherwise.

Hancock's types can be parameterized. Actual parameters are supplied with initializing declarations. Hancock's type system ensures that the types of actual parameters match the declared type of the formal parameters in the type declaration. To provide flexibility, programmers are allowed to omit actual parameters from initializing declarations. When omitted, the runtime system attempts to supply appropriate values, either through a default mechanism or from data stored on disk. If it cannot find appropriate values, then the runtime system halts the program with an error.

The supplied location may be any string-valued expression. The runtime system interprets the value of this expression as a path in the machine's file system. For streams, Hancock allows the special strings `stdin` and `sfstdin` to be used as locations. These strings indicate that the stream will take its data from C's standard input rather than from the file system.

A Hancock program connects data on disk to a variable when it enters a scope that contains an initializing declaration. We use the term *open* to refer to data that has been connected to a variable by an earlier initializing declaration. At that time, in-memory representation associated with the data is initialized from the information on disk, if it exists. If there is no data on-disk, the in-memory representation will be initialized in a type-specific way.

If a Hancock program asks to connect to data that is already open (through another variable), the runtime system checks to ensure that the qualifiers associated with the new request are consistent with the earlier initializing declaration. Two sets of qualifiers are consistent if:

1. the qualifier `const` appears in both sets or neither set, and
2. the qualifier `new` does not appear in the second set.

If the runtime system detects an illegal combination, then the computation will terminate with an error. Otherwise, the new variable will share the same in-memory and on-disk representation with other variables connected to the same data. Hence multiple connections to the same underlying data are aliases.

[1] Currently, values with stream types are read-only; hence we disallow the `new` qualifier for stream declarations.

Chapter 6

Directories

Hancock's directory mechanism provides persistence for statically-sized C types and strings and allows programmers to group related persistent structures. For example, the following code declares a new directory type dir_d that contains myInt, myFloat, myStruct, myArray, and myMap fields.[1]

```
typedef struct {
  int x;
  float y;
} foo_t;

directory dir_d {
  int      myInt default 0;
  float    myFloat default 0.0;
  foo_t    myStruct default {0, 0.0};
  int      myArray[2] default {0,0};
  usage_m  myMap;
};
```

This declaration defines a new type dir_d that can be used to declare variables or function parameters (for example, dir_d d).

Directory fields can have any of the following basic C types:[2]

```
char        signed char          unsigned char
short       unsigned short
int         unsigned int
long        unsigned long
long long   unsigned long long
float       double
char *
```

In the char * case, the field is treated as a string. Directory fields can also have any of the portable types listed in Table 12.1 on page 66. Fields can have C struct types

[1] We use the convention that type names ending in a _d denote directories.

[2] Hancock does not support the type long double.

as long as the types of all the fields in the struct are supported C types. Fields may also have C array types, as long as the array has a statically-known size and the element type of the array is a supported C type.

Fields with standard C types can specify an optional default expression, which Hancock uses to initialize the field when creating a new directory. The type of the default must match the type of the field. For example, the `myInt` field is declared to use zero as a default. In the absence of a default, the value of a field in a new directory is undefined.

In addition to standard C types, directories may contain fields with Hancock map, pickle, or (nested) directory types. Such fields within a directory inherit qualifiers specified in initializing declarations. That is, if a directory is declared to be `const`, then any maps, pickles, or nested directories within that directory will also be declared to be `const`.

6.1 Parameterized directory types

Directory declarations can be parameterized, which reduces the number of distinct types that a programmer must specify without sacrificing the data-safety gained from Hancock's static and dynamic type checking. Such parameters may be used in default expressions and as type parameters for a directory's fields. For example, the following directory declaration takes one integer parameter, `level`.

```
directory cellTower_d(int level) {
  pht_p ctHashTable;
  ct_p  ctab(:level:);
  cellTower_m ctMap(:ctab:);
  char *lastUpdated default "never";
}
```

This parameter is passed as an type parameter to a nested compression table (`ctab`).

Default expressions and actual parameters to nested map, pickle, and directory fields may refer to global identifiers, the formal parameters from the containing directory declaration (*e.g.* the actual parameter for `ctab`), or to previously declared fields (*e.g.* the parameter passed to the map `ctMap`). Hancock initializes directory fields in order, guaranteeing that earlier fields have well-defined values when later ones are initialized. This property means that the `ctab` field will have the appropriate value before the `ctMap` map is initialized. Hancock writes directory fields to disk in the reverse order, which guarantees that all uses of a field will be completed before that field is flushed to disk.

6.2 Representations

Directories are represented in memory as pointers to C `structs`, the definitions of which are generated by the compiler from the directory declarations. For the `dir_d` example, each value is represented as a pointer to a C `struct` defined as follows:

```
struct dirDirectoryRep {
  int      myInt;
  float    myFloat;
  foo_t    myStruct;
  int      myArray[2];
  usage_m  myMap;
};
```

Directories are stored on disk as UNIX directories. Each field of a Hancock directory is represented on disk as a file (or a nested directory) within the corresponding UNIX directory. The files associated with C types contain ASCII representations of the stored data. Fields with Hancock types (maps, pickles, nested directories) use that type's standard representation.

If Hancock's representation is not efficient enough for a particular field, say for a large integer vector, then programmers can use a pickle to customize the representation. This flexibility allows programmers to build prototypes for their applications quickly using Hancock's built-in persistent types, while making it easy to switch to a more efficient representation later.

6.3 Operations

Within Hancock programs, directory fields can be read and written using the standard notations for pointers to structures. For example, the following code fragment first reads from and then writes to the `myInt` field of the directory `d` of type `dir_d`:

```
int x;

x = d->myInt;       /* Arrow notation */
(*d).myInt = x;     /* Star-Dot notation */
```

Data in (non-const) Hancock directories are flushed to disk upon termination of the Hancock program.

Hancock provides a directory copy operation, written `:=:`. The statement

```
new_d :=: d;
```

causes the directory `new_d` to be initialized with data from `d`. The effect of this statement is first to flush to disk the current contents of the in-memory directory `d`. The run-time system then copies the on-disk directory associated with `d` to the on-disk directory associated with `new_d`. Finally, the in-memory representation of `new_d` is initialized from the on-disk directory. The Hancock type system ensures that the two directories have the same type.

Directory assignment, `new_d=d`, is equivalent to pointer assignment; both `new_d` and `d` will point to the same directory.

Chapter 7

Maps

Hancock's map declaration provides a way to associate data with keys. This chapter describes map type declarations, map operations, and map-to-stream conversions.

7.1 Type declarations

The following code declares a new map type, usage_m,[1] that associates values of type uApprox_t with phone numbers:

```
map usage_m {
  key (MINVALIDPN .. MAXVALIDPN-1);
  split (10000, 100);
  value uApprox_t;
  default {0, 0, 0};
};
```

The key clause in this declaration indicates that maps of this type will be indexed by values that range between MINVALIDPN and MAXVALIDPN-1 inclusive. The split clause allows programmers to tune the underlying representation for a particular application. The split clause in this declaration indicates that the key space should be broken into *blocks* of size 10,000 and that these blocks should be further decomposed into *stripes* of size 100. The value clause of a map declaration specifies the type of data to be associated with each key. In this example, the value type is uApprox_t, which was defined on page 16. The default clause specifies a value to be returned when a programmer requests data for a key for which no data has been stored. The default value for a usage_m map is a uApprox_t struct with bucket zero for all three fields. Hancock map declarations also allow the programmer to specify optional compression functions. The remainder of this section describes the various clauses in a map declaration in more detail.

[1] We use the convention that type names ending with _m denote maps.

7.1.1 Keys

Keys typically represent some form of identification number, for example, telephone numbers, credit card numbers, or IP addresses. All keys are represented using the C type `long long`.[2] The `key` clause specifies the range of keys for the map. The example declaration specifies that valid telephone numbers fall between `MINVALIDPN` and `MAXVALIDPN-1` inclusive.

7.1.2 Split

Programmers influence the underlying structure of a map through the `split` specification. The runtime system breaks each key into three pieces: a *block* number, a *stripe* number within that block, and an *entry* number within that stripe. It uses these pieces to index into tables to locate the value associated with each key. Intuitively, blocks serve as the unit of I/O, while stripes serve as the unit of compression. Using the `split` clause, programmers can tune the performance of their applications by controlling the key decomposition for their maps. The `split` clause in the example specifies that there are 10,000 keys per block and 100 keys per stripe for `usage_m` maps.

The best block and stripe sizes for a map depend on the reference patterns of the applications that use the map. The cost of an access is determined by the cost of retrieving the necessary block from disk and the cost of decompressing the appropriate stripe. The decompression cost is proportional to the length of the stripe. As a result, small stripes work well for applications with random access patterns and for applications that have sequential access patterns, but touch only a small percentage of the data in a map. Large stripes work well for applications that have good spatial locality, such as an application that touches all the data in a map using a sequential access pattern.

7.1.3 Values

The value clause specifies the type of the data stored in a map. The only restriction on value types is that they cannot include pointers. The `usage_m` map uses a `view` as its value type, which is a common idiom in Hancock.

We refer to a key with a value in a given map as an *active* key and to an active key and its associated value as an *active item*.

7.1.4 Defaults

The default clause specifies what value should be returned when a programmer requests data for an inactive key. The default may be a constant, as in our earlier example, or a function. A constant default must be an expression that has the map's value type. A function default specifies the name of a function that takes as arguments a key and a pointer to a value of the map type. It stores the default value for the specified key in the space provided by the second argument. Default functions should return `HRS_OK`

[2]Earlier versions of Hancock encoded both the range and split information into the key specification. Hancock now separates this information to simplify the specification and reduce errors.

to signal success and HRS_ERROR to signal an error. For example, the following map declaration

```
int biz_default(long long pn, bLApprox_t *b);

map bizocity_m {
  key (0 .. MAXVALIDPN-1);
  split (10000, 1000);
  value bLApprox_t;
  default biz_default;
};
```

declares the map bizocity_m, which uses the function biz_default, with the given prototype, to compute defaults.

7.1.5 Compression

Hancock maps provide generic compression routines for the values stored in them. For maps with platform-independent value types, the generic compression routines will produce platform-independent on-disk representations. In this case, a map generated on a particular supported platform may be transferred and used on any other supported platform. Chapter 12 describes how to construct platform-independent types.

Hancock's generic routines allow programmers to get started quickly, but they may not be sufficient for a particular application. Programmers often have domain-specific knowledge that allows them to write custom compression routines that are significantly better than the default routines. To exploit this knowledge, map declarations may include optional compress and decompress clauses to specify compression functions. The programmer may specify functions to compress/decompress a single value of the map's value type or to compress/decompress a vector of values.

Single-value compression

The simplest form of user-specified compression allows the programmer to specify a function to compress or decompress a single value. The declaration for usageSV_m gives single-value compression functions:

```
int uSqueeze(uApprox_t *val, uint8 *buffer, int32 bufferSize);
int uUnsqueeze(uint8 *buffer, int32 bufferSize, uApprox_t *val);

map usageSV_m {
  key (MINVALIDPN .. MAXVALIDPN-1);
  split (10000, 100);
  value uApprox_t;
  default {0, 0, 0};
  compress uSqueeze;
  decompress uUnsqueeze;
};
```

A single-value compression function, such as uSqueeze, takes a pointer to a value to be compressed (val), a pointer to a buffer to hold the compressed data (buffer), and

the size of that buffer in bytes as arguments (bufferSize). The function compresses the data, writes it into the buffer, and returns the number of bytes written. If the buffer is not large enough to hold the compressed value, the compression function returns the defined constant HRS_ERROR to signal to the runtime system that an error has occurred.

Similarly, a single-value decompression function, such as uUnsqueeze, performs the reverse operation. It takes a pointer to a buffer containing the compressed data (buffer), the size of that buffer (bufferSize), and space to hold an uncompressed value (val). It reads only as many bytes from the buffer as necessary to reconstitute a value of the appropriate type (uApprox_t). It then stores that value into the space provided (val) and returns the number of bytes consumed from the buffer. If the decompression fails for some reason (*e.g.* the function needed more bytes than were available in the buffer), then the function returns the defined constant HRS_ERROR to signal the failure.

The Hancock runtime system guarantees that the first call to a single-value compression routine receives a buffer that is 8-byte aligned, but does not pad between values to guarantee any particular alignment for subsequent calls. A compression routine that relies on a particular byte alignment must consume buffer space in multiples of that alignment.

If a compression function generates a platform-independent representation, then any map that uses it will be platform-independent as well.

Multiple-value compression

In addition to the single-value form of compression, Hancock supplies a form that allows the programmer to specify functions for compressing and decompressing multiple values. The declaration for usageMV_m gives single-value compression functions:

```
int uVectorSqueeze(uApprox_t *vals, int32 numVals,
                   uint8 *buffer, int bufferSize);
int uVectorUnsqueeze(uint8 *buffer, int32 bufferSize,
                     uApprox_t *vals, int32 numVals);

map usageMV_m {
  key (MINVALIDPN .. MAXVALIDPN-1);
  split (10000, 100);
  value uApprox_t;
  default {0, 0, 0};
  compress uVectorSqueeze;
  decompress uVectorUnsqueeze;
};
```

A vector function, such as uVectorSqueeze, takes a pointer to an array of values to be compressed (vals), the number of values in the vector (numVals), a pointer to a buffer to hold the compressed data (buffer), and the size of that buffer in bytes (bufferSize) as arguments. The function compresses the values, writes them into the buffer, and returns the number of bytes written. If the buffer is not large enough to hold the compressed values, the compression function returns the defined constant HRS_ERROR to signal to the runtime system that an error has occurred.

Similarly, a vector decompression function, such as `uVectorUnsqueeze`, performs the reverse operation. It takes a pointer to a buffer containing the compressed data (`buffer`), the size of that buffer (`bufferSize`), space to hold the uncompressed values (`vals`), and the number of uncompressed values expected (`numVals`). It reconstitutes `numVals` values of the appropriate type (`uApprox_t`) and should consume the whole buffer in the process. It then stores the values into the space provided (`vals`) and returns the number of bytes consumed. If the decompression fails for some reason (*e.g.* the function needed more bytes than were available in the buffer), then the function returns the defined constant `HRS_ERROR` to signal the failure.

With the exception of a few calls to compress information used to help identify the map, vector compression functions will be called once per stripe. The runtime system packs the active items in a stripe into a vector and then calls the vector compression function (or alternatively, calls the vector decompression function and then transfers the decompressed active items into the stripe).

As with single-entry compression, if a compression function generates a platform-independent representation, then any map that uses it will be platform-independent as well.

7.2 Parameterized type declarations

Maps may be parameterized, which reduces the number map types that a programmer needs to specify. Type parameters can be used in expressions in the `key`, `split`, `default`, and compression clauses. Default and compression functions can be specified using type parameters, but such parameters must be specified will all initializing declarations that use the type. Hancock can extract some values, such as the key ranges and the split values from self-identification information stored in the on-disk representation of a map, but it cannot extract unspecified functions.

Parameters can be passed as arguments to function defaults and compression functions. For example, the bizocity map declared earlier constructs defaults by querying another data source. The map declaration in Figure 7.1 redefines `bizocity_m` to use a type parameter, `kv` of type `knownValues_t`, that is passed to the biz_default function. The earlier definition of `biz_default` would have used a global variable for the known values table. This version allows different `bizocity_m` maps to use different values for the back-up data structure.

Because programmers are not required to supply parameters when declaring a map variable, functions that use type parameters, such as `biz_default`, need to know whether the programmer supplied the parameters for a particular map variable. Hancock passes this information to the function with a flag (passed as the first parameter to the function) that indicates whether parameters were supplied. Default (and compression) functions are responsible for taking appropriate action if the `set` flag indicates that no parameters were supplied at initialization time. Such actions might include constructing an appropriate value or returning an error.

Another common use of type parameters is to pass a compression table to a map. For example, the `cellTower_m` map declaration in Figure 7.2 takes a compression table as a parameter, which it passes as an argument to the compression and decom-

```
int lookup(knownValues_t kv, long long pn, bLApprox_t *b);

int biz_default(char set, knownValues_t kv,
                long long pn, bLApprox_t *b) {
  if (set)
    return lookup(kv, pn, b);
  else {
    b->in = BIZ_DEFAULT;
    b->out = BIZ_DEFAULT;
    return HRS_OK;
  }
}

map bizocity_m(knownValues_t kv) {
  key (0 .. MAXVALIDPN-1);
  split (10000, 1000);
  value bLApprox_t;
  default biz_default(:kv:);
};
```

Figure 7.1: Map declaration illustrating use of type parameters to pass auxiliary information to default functions.

```
#include "ctp.hh"
int ctSqueeze(char set, ct_p ctab, profile_t *from,
              uint8 *buffer, int32 bufferSize);

int ctUnsqueeze(char set, ct_p ctab, uint8 *from,
                int32 bufferSize, profile_t *to);

#define CTM_DEFAULT  {{CTD, CTD, CTD, CTD, CTD}, \
                      {0.0, 0.0, 0.0, 0.0, 0.0}, \
                      0.0}

map cellTower_m(ct_p ctab)  {
  key (MINPN .. MAXPN);
  split (10000, 100);
  value profile_t;
  default CTM_DEFAULT;
  compress ctSqueeze(:ctab:);
  decompress ctUnsqueeze(:ctab:);
};
```

Figure 7.2: Map declaration illustrating use of type parameters to pass compression table to compression functions.

pression functions. The C function `ctSqueeze` a set parameter and compression table (implemented as a pickle type) plus a pointer to a `profile_t` and returns a collection of bytes (`buffer` and `bufferSize`) and the size of that collection (the return value).

7.3 Operations

Hancock provides five operations for manipulating items in maps: read, write, remove, test key, and test active key. Hancock's indexing operator, written `<: ... :>`, can be used as an r-value (for reading) or as an l-value (for writing). Hancock's map remove operator, written `\<:..:>`, removes a key/value pair from the map. Hancock's test key operator, written `?<: ... :>`, queries a map to determine whether the key is in the map's range. Finally, Hancock's test active key operator, written `@<: ... :>`, queries a map to determine whether the key is active. It is a run-time error to pass an out-of-range key to any of these operations other than test key.

The following code first reads the value for the key `pn` from map `u`, writes a different value `a1` into map `u` for the key `pn`, tests whether the key `pn1` has a programmer-set value in map `u`, and if so removes that key/value pair from the map.

```
usage_m u = "98216.u";
long long pn, pn1;
uApprox_t a, a1;

a = u<:pn:>;                   /* Read pn's data    */
  ...
a1 = ...;

u<:pn:> = a1;                  /* Write pn's data   */

if (u?<:pn:> && u@<:pn1:>)     /* Is pn1 in u's range and if
                                * so it have data?  */
  u\<:pn1:>;                   /* Remove pn1's data */
```

A common Hancock idiom uses the indexing operator to extract an approximate value from a map and the view operator to convert that value into the corresponding "exact" view. For example,

```
usage_m u;
uSig_t s;
long long pn;

s = u<:pn:>$uSig_t;      /* Read pn's data,
                          * convert to uSig_t view */
  ...
u<:pn:> = s$uApprox_t;  /* Convert to uApprox_t view,
                          * write pn's data        */
```

Hancock also provides a map copy operator, written `:=:`. The statement `new_u:=:u` causes the map `new_u` to be initialized with the data from the map `u`. The types of

new_u and u must be the same. The compiler checks this property statically and the runtime system checks it dynamically as well. In addition, the runtime system checks that map new_u is empty. If it is not, the runtime system will raise an error and exit.

Map assignment, new_u=u, is equivalent to pointer assignment; both new_u and u will point to the same map.

7.4 Stream conversion

Hancock allows maps to be converted into streams so that programmers can perform an operation for every active key. The expression myMap[s..f] generates a stream of keys that fall between the value of the bounds expressions s and f inclusive for which myMap has programmer-set values. Identifier myMap names the map to be converted into a stream. The expressions s and f bound the portion of the map to be converted. The expression myMap without the syntax [s..f] generates a stream that contains all active keys in the map. We describe stream operations in general in Chapter 9. Section 9.3 describes the context in which map-to-stream operations may be used. In addition, we describe a collection of library functions that can be used to generate more complex streams from maps in Chapter 11.

Chapter 8

Pickles

Hancock's `pickle` declaration provides a way for users to specify their own persistent data representations. Pickles can also be used to implement data structures that keep in memory only a (potentially small) portion of the data on disk.

8.1 Type Declarations

The following declaration introduces a new pickle type, `pht_p`, that implements a persistent hash table:[1]

```
pickle pht_p {init_pht => pht_rep => flush_pht};
```

The declaration starts with the `pickle` keyword and specifies four things: the name of the pickle type (`pht_p`), the name of a function for opening the pickle (`open_pht`), a C-type specifying the in-memory representation (`pht_rep`), and the name of a function for flushing the pickle to disk/deallocating any resources associated with the pickle (`flush_pht`).

The open function initializes the in-memory representation from the contents of a file passed to it as an argument. If that file is empty or non-existent, then the function generates appropriate initial values. The type of this function must match the prototype:

```
int init_pht(Sfio_t *fp, pht_rep *data, char readonly);
```

The first parameter is the file pointer; the Hancock run-time system opens the corresponding file before calling the function `open_pht`. The second parameter is a pointer to the in-memory representation of the pickle. The third parameter is a flag that indicates whether or not the pickle was declared to be read only. The return value signals success or failure of the function. A return value of `HRS_OK` signals success, where as a return value of `HRS_ERROR` signals an error. This function should not close the file `fp`.

The type of the flush function must match the prototype:

[1] We use the convention that type names ending with an `_p` denote pickles.

```
int flush_pht(Sfio_t *fp, pht_rep *data, char close);
```

The first parameter is the same file pointer that the run-time system passed to the pickle open function. The run-time system does not close the file after the call to the open function. The second parameter points to the in-memory representation of the pickle. If the `close` flag is true, the function should free any resources it has allocated. The function should not close the file `fp`. The Hancock runtime system will call this function even if the associated pickle is opened in read-only mode so that the pickle can deallocate its resources. In this case, the flush function should not attempt to write to the file, as a write operation may cause an error if the user does not have the write permission for the on-disk file. Currently, the flush function does not take as an argument a `readonly` flag; hence the pickle designer should cache this information in the in-memory representation of the pickle during the initialization function.

8.2 Example: Persistent hash table

To illustrate pickles, we define a persistent hash table using a pickle. The on-disk representation for our persistent hash table implementation is a file containing one line per hash table entry. Each line has the format:

```
hash:len:string
```

where `string` is the value in the hash table, `len` is its length, and `hash` is its hash code. For the in-memory representation, we use a standard hash table implementation, the interface of which appears in Figure 8.2, augmented with two extra fields:

```
typedef struct {
  hash_table *ht;
  char readonly;
  char updated;
} pht_rep;
```

These extra fields are flags that indicate whether the current hash table is read-only and whether the in-memory representation has been updated. In addition to selecting data representations, we need to define initialization and flushing functions that convert between the on-disk and in-memory representations of the hash table. Code for suitable functions appears in Figures 8.2 and 8.3.

The initialization function, `init_pht`, takes a file pointer, a pointer to space to hold the in-memory representation of the persistent hash table, and a flag to indicate whether the hash table should be read-only. The function initializes the hash table and sets the `readonly` flag from the function parameter. It then reads the strings and their associated hash values from the file one at a time, inserting the resulting pairs into the hash table. If an error occurs within the hash table, the initialization function returns `HRS_ERROR`. Otherwise, the function returns `HRS_OK`.

The flush function, `flush_pht`, also takes a file pointer and a pointer to the in-memory representation. If the persistent hash table is not `readonly` and it has been `updated`, the function flushes the contents of the in-memory hash table to the file. It truncates the file after the last write in case the hash table shrunk in size. Finally, if the

```
/* Trivial linear hashing for strings. */
#define HASH_TABLE_SIZE 100001

#define HASH_OK 0
#define HASH_FULL -1
#define HASH_ERROR -1
#define HASH_ITER_FOUND 1
#define HASH_ITER_DONE 0
#define HASH_DEFAULT 0

typedef char * hash_value;

typedef struct {
  hash_value data[HASH_TABLE_SIZE];
} hash_table;

/* Create new, empty hash-table. */
hash_table * hash_empty();

/* Close hash table */
void hash_close(hash_table *s);

/* Insert copy of v in table, return hash code as out parameter.
 * If v already in table, returns existing hash.
 * Return HASH_OK, HASH_ERROR or HASH_FULL. */
int hash_insert(hash_table *s, hash_value v, unsigned int *hash);

/* Insert copy of v in table with hash code hash.
 * Return HASH_OK, HASH_ERROR or HASH_FULL. */
int hash_insert_pair(hash_table *s, hash_value v, unsigned int hash);

/* Lookup hash in table, returns HASH_DEFAULT if no value present. */
hash_value hash_lookup(hash_table *s, unsigned int hash);

/* Iterator type for walking over hash table. */
typedef int* hash_iter;

/* Create new iterator. */
hash_iter hash_init_iter(hash_table *s);

/* Get next hash value, hash code pair.
 * Non-copy semantics for hash value.
 * Return HASH_ITER_FOUND or HASH_ITER_DONE */
int hash_next(hash_table *s, hash_iter i,
              hash_value *v, unsigned int *hash);

/* Close iterator. */
void hash_close_iter(hash_table *s, hash_iter i);
```

Figure 8.1: Interface to a string hash-code library.

```
/* File representation:
 * hash0:len:string_0
 * hash1:len:string_1
 * ...
 * hashn:len:string_n
 */

int init_pht(Sfio_t *fp, pht_rep *data, char readonly){
  int result, len;
  unsigned int hash;
  char *s;
  data->ht = hash_empty();
  data->readonly = readonly;
  data->updated = 0;
  if (0 == sfsize(fp)) return HRS_OK; /* empty file, okay */
  else
    while (!sfeof(fp)) {
      if (2 != sfscanf(fp, "%u:%d:", &hash, &len))
        return HRS_ERROR;
      s = (char *)malloc(len+1);
      if (NULL == s) return HRS_ERROR;
      if (1 != sfscanf(fp, "%s\n", s)) return HRS_ERROR;
      if (HASH_OK != hash_insert_pair(data->ht, s, hash))
        return HRS_ERROR;
    }
  return HRS_OK; /* okay */
}
```

Figure 8.2: Initialization function for persistent hash tables.

```
int flush_pht(Sfio_t *fp, pht_rep *data, char close){
  /* add error checking. */
  int len;
  char *s;
  unsigned int hash;
  hash_iter i;

  if ((!data->readonly) && (data->updated)) {
    Sfoff_t curLoc;
    sfseek(fp, (Sfoff_t) 0, SEEK_SET);
    i = init_iter_pht(data);
    while (HASH_ITER_FOUND == next_iter_pht(data, i, &s, &hash)) {
      sfprintf(fp, "%u:%d:", hash, strlen(s));
      sfprintf(fp, "%s\n", s);
    }
    close_iter_pht(data, i);
    curLoc = sfseek(fp, (Sfoff_t) 0, SEEK_CUR);
    ftruncate(sffileno(fp), curLoc);
  }
  if (close)
    hash_close(data->ht);
  return HRS_OK; /* okay */
}
```

Figure 8.3: Flush function for persistent hash tables.

```
typedef struct {
    ...
} ct_rep;

int init_ct(char set, int level,
            Sfio_t *fp, ct_rep *data, char readonly);
int flush_ct(Sfio_t *fp, ct_rep *data, char close);

pickle ct_p(int level) {init_ct(:level:) => ct_rep => flush_ct};
```

Figure 8.4: Compression table pickle type definition.

```
#define COMPRESSION_LEVEL_DEFAULT   6
int init_ct(char set, int level, Sfio_t *fp,
            ct_rep *data, char readonly){
  data->readonly = readonly;
  if (0 == sfsize(fp)) {     /* empty file */
    if (set)                        /* Was parameter given? */
      data->level = level;          /* Yes: Use parameter */
    else                            /* No:  Use default */
      data->level = COMPRESSION_LEVEL_DEFAULT;
    /* Construct empty compression table */
  } else {
    /* Intialize compression table from contents of file. */
  }
  return HRS_ERROR; /* Couldn't initialize compression table. */
}
```

Figure 8.5: Compression table pickle initialization function.

`close` flag is set, the flush function deallocates the resources held by the persistent hash table.

8.3 Type parameters

Our example pickle type, `pht_p`, does not take any parameters, but pickles, like directories and maps, may be parameterized. Figure 8.4 contains the type definition for a compression table pickle that takes an integer argument (`level`) for specifying how hard the compressor should work to save space. The initialization function `initCTab`, which appears in Figure 8.5, takes the usual pickle initialization function parameters (`fp`, `data`, and `readOnly`) plus two extra parameters (`set` and `level`) that arise from the `level` type parameter. The `set` parameter indicates whether the parameter was supplied when a given pickle variable was declared. The `level` parameter contains the supplied value if one exists or an undefined value otherwise. The write function takes only the usual parameters because it was not declared to take any type parameters.

```
int insert_pht(pht_p data, char *s, unsigned int *h){
  if (!data->readonly){
    data->updated = 1;
    return hash_insert(data->ht, s, h);
  } else
    return PHT_ERROR;
}
```

Figure 8.6: Insert function for persistent hash tables.

8.4 Operations

Hancock treats values with pickle types as pointers to the underlying representation type. For example, once a persistent hash table has been opened, the programmer can query the table and insert new values using the function `insert_pht` shown in Figure 8.6.

Hancock provides a pickle copy operation, written using the `:=:` notation. The statement `new_p :=: p` opens the pickle `new_p` with the data from `p`. In more detail, the Hancock run-time system synchronizes the in-memory and on-disk representations of `p` by calling `p`'s flush function with the `close` flag set to false. The run-time system then copies the file associated with `p` to the file associated with `new_p`. It then initializes the in-memory representation of `new_p` using the pickle's open function. The program must have connected both pickles to on-disk files prior to the copy statement. The Hancock type system ensures that the two pickles have the same type and hence the same open and writing functions.

Pickle assignment, `new_p = p`, is equivalent to pointer assignment; both `new_p` and `p` will point to the same pickle.

8.5 Advanced usage

As mentioned at the beginning of this section, pickles enable users to keep in memory only a portion of a persistent data-structure. To achieve this effect, the open function for the pickle should store the file pointer in the in-memory representation. Because the Hancock run-time system does not close the file pointer, it continues to provide access to the data on disk throughout the life-time of the pickle. This technique allows users to cache only a portion of the persistent data in memory, improving response times for queries involving frequently touched data.

Persistent hash tables, such as those discussed in Section 8.2, provide a mechanism for processing variable-width data, such as strings for identifying nodes in network or URLs. When consuming a stream (see Chapter 9), the variable-width data is hashed and the associated fixed-width values are placed in the logical stream. These associated values may be used to index into maps, so we can build profiles of nodes identified by strings or URLs. Using directories (see Chapter 6), we can associate the persistent hash table with the maps on disk, so that the linkage between the two data sources is evident.

Chapter 9

Streams

The `stream` construct provides sequential access to collections of records. This chapter describes how to read stream descriptions, consume streams, and write new stream descriptions. It also discusses how streams are represented on disk. For expository purposes, we discuss how to read stream descriptions before describing how to write such descriptions.

9.1 Understanding stream descriptions

Hancock's stream mechanism allows the records in a stream to have two representations: a *physical* representation, which describes how the records appear on disk, and a *logical* representation, which describes how they appear in memory. It is often the case that the physical representation is highly encoded, while the logical representation is more convenient to manipulate. A Hancock `stream` declaration specifies how to convert the records in a stream from their physical to their logical representation. Not all records in a stream may be valid, so the conversion from the physical representation to the logical representation should also validate the physical record and reject invalid records. Clients of a given stream definition use the logical representation; only the code that describes the translation manipulates the physical representation. Such clients can assume that the logical record is valid, which simplifies downstream code.

Hancock provides three kinds of stream declarations: a general form for consuming data stored in files in any kind of format, a specialized form for streams whose physical representation is a sequence of binary records, and a generator form for creating streams that have no on-disk representation. As with Hancock's other persistent types, stream declarations may take parameters.

The following code specifies a general stream of records that log information about customer connections to the WorldNet ISP.

```
stream worldNet_s {
  getValidSession: Sfio_t => session_t;
};
```

This declaration defines a new stream type `worldNet_s`.[1] It specifies that the logical representation for this stream is the type `session_t`, given after the "=>" notation. Values with type `session_t` (defined in Appendix D) log information particular to a single WorldNet session. The identifier `getValidSession` names a function that reads an ASCII representation of a WorldNet record from its file argument and translates this physical representation to a logical one. The definition of `getValidSession` appears in Figure 9.5 on page 54.

As noted earlier in Chapter 5, stream type definitions may be parameterized. As an example of such a stream, the following declaration parameterizes the general stream `worldNetBnd_s` by two `int`s:

```
stream worldNetBnd_s(int min, int max) {
  getValidSessionBnd(:min, max:) : Sfio_t => session_t;
}
```

The translation function `getValidSessionBnd` uses these parameters to remove from the stream all records containing customer identifiers out of the range `min` to `max`, the lowest and highest legal customer identifiers for a given application.

Binary streams are similar to general streams except binary streams specify a translation function that maps from the physical representation to the logical without referring to the containing file. In this case, the physical representation is specified as a C struct, and the containing file is a sequence of such structs in binary form. The Hancock run-time system manages file operations for binary streams. As an example, the following declaration defines a binary stream of call-detail records:

```
stream callDetail_s {
  getValidCall : pScampRec_t => scampRec_t;
};
```

The identifier `getValidCall` names a translation function that maps the physical record type `pScampRec_t` to the logical type `scampRec_t`.

Generator streams do not have a physical representation. Instead, the translation function generates values, often using type parameters to store state. As an example, the code

```
stream generateInt_s(int *i) {
  nextElem(:i:) : void => int;
};
```

declares a generative stream `generateInt_s` that takes as a parameter a pointer to an integer. The function `nextElem` uses this parameter to compute the next integer and to store the state of the generator.

9.2 Representation

Each (non-generative) stream is represented on disk either as a file of physical records or as a directory containing files of physical records. If the stream is represented on disk as a directory, Hancock assumes that all the files in the directory belong to that

[1] We use the convention that type names ending in a `_s` denote `streams`.

stream. To connect a stream on disk with a Hancock variable, we must associate the name of the file or directory containing the stream data with a variable, using either an initializing declaration (Chapter 5) or a sig_main parameter (Chapter 10).

A common error with streams represented on disk as UNIX directories is to leave extra files in the directories. The Hancock run-time system assumes these files belong to the associated stream and attempts to process them, often generating incorrect results. Files annotated with # and ~ are particularly pernicious as they often contain duplicates of legitimate data, causing the program to terminate normally but with incorrect values.

9.3 Consuming streams

Hancock's iterate statement provides a way to process streams. As a running example, Figure 9.1 shows a fragment of Hancock code that uses the iterate statement to compute the number of seconds each phone number in a call stream was active. Each iterate statement contains header clauses that specify an initial stream, a filtering transformation, a sorting transformation, and a function to detect events in the transformed stream. These clauses may be written in any order, but they are executed in a fixed order determined by Hancock. The body of the iterate statement contains a list of event clauses that specify how to respond to detected events. In this section, we explain each of these pieces by referring to the example in Figure 9.1.

The over clause of the iterate statement specifies an initial stream to transform. In this case, the variable calls names a stream of type callDetail_s. In addition to iterating over stream variables, we may use the iterate statement in conjunction with a map-to-stream expression (see Section 7.4) to iterate over the keys in a map that have user-supplied data associated with them. The over clause is required.

The filteredby clause specifies a predicate to remove unwanted records from the stream. This predicate is applied to each successive record in the stream. Records for which the predicate is true are retained. The programmer may specify this predicate in two ways: as a function identifier or as an in-line expression. In the example, the identifier onlyComplete names a function that returns true when passed a pointer to a call-detail record (*rp) for a completed call. In the expression syntax, the filteredby clause may be written:

```
filteredby (rp) !rp ->isIncomplete
```

In this notation, the parenthesized variable rp is a formal parameter, bound at run-time to a pointer to the next logical record in the stream. The boolean expression that follows indicates which calls to retain, complete calls in this case. Currently, only the variable bound in the filteredby clause (rp) and global identifiers are in scope in the filteredby expression. The filteredby clause is optional. If given, filtering precedes sorting.

The sortedby clause describes a sorting transform for the filtered stream by listing fields from the stream's logical record type that constitute the sort key. In the example, the identifier origin indicates that the records from stream calls should be sorted by the originating telephone number. The clause sortedby origin,

```
#include <stdio.h>

#define LAMBDA .15

double blend(double oldVal, double newVal){
  return oldVal * (1-LAMBDA)  + newVal * LAMBDA;
}

int onlyComplete(scampRec_t *r)
{ return !r->isIncomplete; }

void out(callDetail_s calls, usage_m usage)
{  uSig_t u;
   uSig_t cumusage;

  iterate
    ( over calls
      filteredby onlyComplete
      sortedby origin
      withevents originDetect ) {

    event npa_begin(areacode_t npa) {
      printf("%d\n", npa);
    }

    event line_begin(pn_t pn) {
      u = usage<:pn:>$uSig_t;
      cumusage.out = 0;
      cumusage.outTF = 0;
    }

    event call(scampRec_t r) {
      if (r.isTollFree)
         cumusage.outTF += (double)r.duration;
      else
         cumusage.out +=  (double)r.duration;
    }

    event line_end(pn_t pn) {
      u.out = blend(u.out, cumusage.out);
      u.outTF = blend(u.outTF, cumusage.outTF);

      usage<:pn:> = u$uApprox_t;
    }
  };
}
```

Figure 9.1: Iterate statement from the `usage` signature.

connecttime would yield a stream sorted primarily by the originating telephone number and secondarily by the values in the connecttime field. The values in the key are compared using the ordering associated with their type, with the smallest value appearing first in the resulting stream. The ordering for basic C types corresponds to the semantic ordering for those types. Values with struct types are compared field by field, in the order that the fields are specified in the struct, using the appropriate semantics for each field (that is, a field with the type float will use floating point comparison semantics for that field). Fixed-width arrays are sorted lexicographically using the appropriate semantics for comparing array elements. If the list in the sortedby clause contains fields that are not in the stream's logical record type, the compiler generates a type error.

If the logical representation for a stream is not a struct, or if one wishes to use the entirety of a logical record as a key, one can use the sorted clause as an alternative to the sortedby clause. The sorted clause takes no parameters and transforms the stream so that it is sorted in the order determined by the logical record type. Both the sorted and sortedby clauses are optional, but at most one may be specified in any iterate statement. Sorting precedes filtering.

The withevents clause specifies how to detect events in the (filtered, sorted) stream. As with filteredby, the withevents clause takes two forms: a function identifier or an in-line expression. In the first case, the identifier names a function to detect events in the stream. In the second case, the expression computes the events in-line. In either case, the event detection code is applied once for each record in the (filtered, sorted) stream. This code takes as a parameter a *window* (Section 9.4.1) into the stream and returns a value with a Hancock *multi-union* type (Chapter 4). The next section describes the Hancock window type and how to specify event detection code.

In the example, the function originDetect detects changes in the originating telephone number in successive records in the call stream by returning values with type:

```
munion line_e {: areacode_t npa_begin,
                 exchange_t nxx_begin,
                 pn_t line_begin,
                 scampRec_t call,
                 pn_t line_end,
                 exchange_t nxx_end,
                 areacode_t npa_end :};
```

When the originating number of the current record has a different area code than the originating number of the *previous* record in the stream, function originDetect raises the npa_begin event and associates the area code of the current originating telephone number with that event. In this situation, the function also raises the nxx_begin and line_begin events, because those values logically change whenever the area code changes. These events carry the exchange and line for the current originating phone number, respectively. If the originating number of the current record has a different area code than the originating number of the *next* record, originDetect raises the npa_end event and associates the area code of the current originating number with that event. In this situation, the function also raises the nxx_end and line_end events. Function originDetect treats originating phone numbers with different exchanges and lines analogously. It raises the call event for

```
#include <stdio.h>

void count(usage_m usage)
{ int count = 0;
  iterate
    ( over usage ) {
      event (long long * pn) {
        count++;
      }
  };
  printf("Count = %d\n", count);
}
```

Figure 9.2: Iterate statement without a `withevents` clause.

every record in the call stream, with the entire record as the associated value.

The `withevents` clause is optional. When omitted, Hancock uses an internal event-detection function that returns a multi-union with a single (anonymous) label that carries as a value a pointer to the logical record type of the stream.

The body of the `iterate` statement consists of a list of event clauses. These clauses specify the code to execute when the event detection function triggers events. Each event clause has a name,[2] a typed parameter, and a body of arbitrary C/Hancock code. The name of each event clause must be one of the labels in the multi-union returned by the event detection function, and the type of the parameter to the clause must match the type associated with the corresponding multi-union label.

As an example, the code in the `npa_begin` clause runs whenever the multi-union returned by the `originDetect` function contains the `npa_begin` label. The parameter to this clause, `npa`, is set to the area code that the `originDetect` function associated with the `npa_begin` event. The body of this clause prints out the associated area code for logging purposes. The `line_begin` event clause runs whenever `originDetect` triggers a `line_begin` event, and the parameter `pn` is initialized to the phone number associated with that event. Event clauses `call` and `line_end` are analogous.

When the event detection function returns a multi-union with more than one event, the event clauses are executed in the order that they appear in the *body of the `iterate` statement*. For example, if function `originDetect` raised both a `line_begin` and an `call` event for a particular record, then the `line_begin` event code would run followed by the `call` event code.

When the event detection function is omitted, there should be only one event clause. This clause should be *anonymous*, *i.e.*, its name should be omitted. The type of its parameter should be a pointer to the logical record type of the stream. The code in Figure 9.2 uses this form of iterate to count the number of entries in a map.

[2]Except when the `withevents` clause in omitted, in which case there can be at most one event clause.

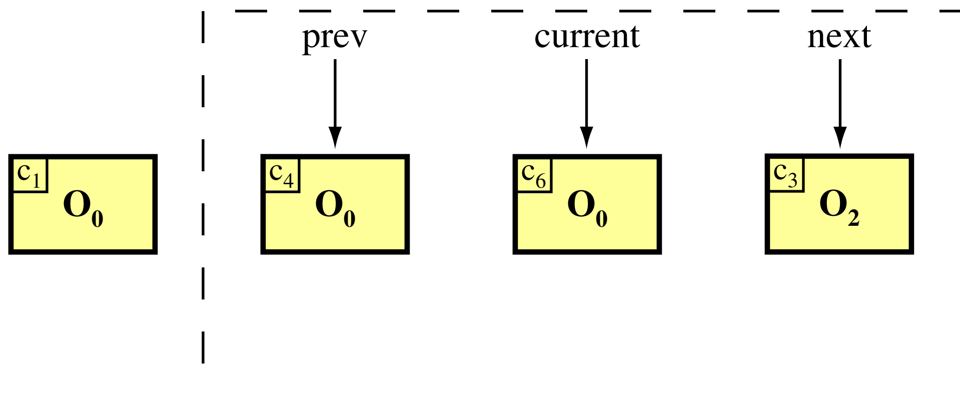


Figure 9.3: Sample stream window.

9.4 Event detection

As mentioned in the previous section, event detection code examines a small window of a stream and returns a multi-union describing the events detected in that window. This section introduces *windows* and illustrates both event-detection functions and in-line event-detection expressions.

9.4.1 Windows

Hancock provides a type, called a *window*, that gives a programmer a view onto a small contiguous part of a stream. The size of the window determines how many stream records can be viewed at one time. A window is like an array, but has the added notion of a "current" element. In specifying a window, the programmer indicates the placement of the current record in the window. For example, the declaration `pn_t *w[3:1]` specifies that `w` is a window of size three onto a stream with records of type `pn_t`. A pointer to the current record appears in the middle slot of the window, *i.e.*, in `w[1]`. Slots with lower indices (`w[0]`) store pointers to records earlier in the stream; slots with higher indices (`w[2]`) look ahead to records appearing later in the stream. If the window overlaps either the beginning or the end of the stream (or both), the slots with no corresponding stream record are set to NULL.

Figure 9.3 shows a sample window into a stream. (These calls may seem out of order, but recall that they were sorted by the originating number prior to event detection.) Calls c_4, c_6, and c_3 make up the window, with c_6 being the "current" record.

9.4.2 Example detection function

For a stream of calls of type `scampRec_t`, the `originDetect` function uses a window of size three with a current position of one to detect the start of an originating telephone number in the stream and to detect the end of an originating telephone number in the stream. It returns a `line_e` multi-union. The code for `originDetect`

and two auxiliary functions (`beginLineDetect` and `endLineDetect`) is shown in Figure 9.4.

9.4.3 Example in-line detection code

For a `callDetail_s` stream `s`, the following event detection code triggers a `line` event carrying the originating phone number for each record in the stream:

```
munion origin_e {: long long origin :};
void doIter(callDetail_s s){
  iterate
    ( over s
      withevents (w[1:0]) {: origin = w[0]->origin :} ) {

      event origin(pn_t origin){
                ...
      }
  }
}
```

The variable `w` is bound to a window of size 1 with current offset of zero in the stream `s`. Variable `w` is in scope in the following expression,

```
{: origin=w[0]->origin :}
```

which uses this value to calculate the desired multi-union.

Another use for in-line events is to allow event detection functions to take extra arguments. For example, the event detection expression in the following code:

```
#define ORIGIN    1
line_e callDetect(scampRec_t *w[3:1], char type);

void doOrigin(callDetail_s calls, usage_m u){
  iterate(over calls
          filteredby onlyComplete
          withevents (w[3:1]) callDetect(w,ORIGIN)  ){
          ...
  }
}
```

passes the window `w` and a flag indicating which of the phone numbers in the logical record should be used to generate events to the function `callDetect`. This auxiliary function then generates a multi-union describing the events detected for the window. Unlike the `filteredby` clause, local variables and formal parameters are in scope for the `withevents` clause.

9.5 Writing stream descriptions

In this section, we describe how to define data sources for use within Hancock programs. Defining a data source involves the following steps:

```
line_e beginLineDetect(pn_t *prev, pn_t *current) {
  areacode_t a = extractNPA(*current);
  exchange_t n = extractNPANXX(*current);

  if ((prev==NULL) || (extractNPA(*prev) != a))
    return {: npa_begin = a, nxx_begin = n, line_begin = *current :};
  if (extractNPANXX(*prev) != n)
    return {: nxx_begin = n, line_begin = *current :};
  if (*prev != *current)
    return {: line_begin = *current :};

  return (line_e) {: :};
}

line_e endLineDetect(pn_t *current, pn_t *next){
 areacode_t a = extractNPA(*current);
 exchange_t n = extractNPANXX(*current);

  if ((next == NULL) || (a != extractNPA(*next)))
    return {: line_end = *current, nxx_end = n , npa_end = a :};
  if (n != extractNPANXX(*next))
    return {: line_end = *current, nxx_end = n :};
  if (*current != *next)
    return {: line_end = *current :};

  return (line_e) {: :};
}

line_e originDetect(scampRec_t *w[3:1])
{ line_e b,e;
  pn_t *prev, *curr, *next;

  prev = ((w[0] == NULL) ? NULL : &(w[0]->origin));
  curr = &(w[1]->origin);
  next = ((w[2] == NULL) ? NULL : &(w[2]->origin));
  b = beginLineDetect(prev,curr);
  e = endLineDetect(curr,next);

  return b :+: {: call = *w[1] :} :+: e;
}
```

Figure 9.4: Sample event detection function.

1. understanding the physical representation of the data on disk, including what constitutes a “legal” record in the stream,
2. designing a general, logical representation of the data suitable for computation,
3. writing a translation function that maps the physical to the logical representation, and then
4. using these pieces to write a stream declaration.

9.5.1 Runtime model for consuming a stream

When iterating over a given binary or general stream, the runtime system monitors the physical data associated with the stream, which can be stored either all in one file (if the path given in the initializing declaration for the stream was a file) or in a collection of files (if the path was a directory). While physical data remains, the runtime system repeatedly invokes the translation function for the stream to produce logical records. Iteration terminates when the physical data is exhausted.

The translation function for a stream is responsible for producing a logical record as an out parameter and returning a status flag. This flag can have one of four values:

`HRS_STREAM_KEEP_REC`	logical record is valid
`HRS_STREAM_DROP_REC`	logical record is invalid, discard
`HRS_STREAM_STOP`	terminate iteration
`HRS_STREAM_ERROR`	catastrophic error detected, terminate program

The first indicates that the logical record returned by the translation function is valid and should be seen in the logical stream. The second indicates that the translation function detected a problem with the current physical record and could not produce a logical record from it. The runtime system will ignore the invalid logical record and invoke the translation function again. The third flag `HRS_STREAM_STOP` indicates that the translation function detected the end of the logical stream (even though there may be more physical data). This flag causes iteration to terminate. Finally, `HRS_STREAM_ERROR` signals that the translation function detected a catastrophic failure in the data. It instructs the runtime system to halt program execution immediately.

When iterating over a generative stream, the runtime system repeatedly invokes the translation function until the translation function returns `HRS_STREAM_STOP` or `HRS_STREAM_ERROR`.

9.5.2 Declaration forms

As described earlier, Hancock’s stream declaration has three variants: general, binary, and generative. In the general case, the data may be stored in any format; the person who defines the stream type is responsible for reading the data from disk given a file pointer. The binary form, in contrast, requires that the stream source contains fixed-width binary records, in which case Hancock manages the reading of the binary records from disk. Generative streams allow programmers to create streams at runtime. Through the use of type parameters, programmers can use generative streams to convert other sources of data, such as pickles or other data structures, into streams.

All three variants share the following declaration form:

```
stream stream_s (ty_1 p1, ...,  ty_n pn) {
      translationFn(:e1, ..., em:) : PhysRep_t => LogRep_t;
};
```

where `stream_s` is the name of the new stream type, the `pi` are (optional) parameters that are in-scope for the remainder of the declaration, identifier `translationFn` is the translation function, the expressions `ej` are (optional) parameters to the translation function, `PhysRep_t` is a C type describing the physical representation of the stream, and `LogRep_t` is a fixed-width C type describing the logical representation. The form of `PhysRep_t` determines the type of stream as indicated by the following table:

`Sfio_t`	General stream
`void`	Generative stream
Any other fixed-width C type	Binary stream

The type of the translation function is determined by the stream declaration. If the declaration passes parameters to the function, *i.e.*, the `ej` list is non-empty, the function's first parameter must be a character flag, which is set when the initializing declaration provides values for the type parameters (the `pi`). If the function does not take parameters, the character flag is omitted. The function then has a sequence of formal parameters corresponding to the actual parameters supplied in the stream declaration. In the example, `translateFn` would have `m` such parameters. The type of the `j`th parameter must correspond to the type of `ej`. The next parameter depends on the value of `PhysRep_t`. If `PhysRep_t` is non-`void`, the next parameter has type `PhysRep_t *`. It is through this parameter that the function accesses the physical representation of the stream. If `PhysRep_t` is `void`, indicating a generative stream, this parameter is omitted from the translation function. The last parameter to the translation function has type `LogRep_t *`. The translation function uses this parameter to return valid logical records. The return type of the function is `int`, which is used as a flag to return status information as described in Section 9.5.1.

In the next two subsections, we provide example stream definitions. The first uses a general stream to describe session logs from the WorldNet ISP. The second describes how to use generative streams to create a stream of keys stored in a persistent hash table.

9.5.3 Example: WorldNet session logs

The WorldNet session data comes in an ASCII format, so we will use a general stream declaration to describe it:

```
stream worldNet_s {
  getValidSession: Sfio_t => session_t;
};
```

The programmer is responsible for understanding the physical format of the data, designing a logical representation for the data, and finally for writing a translation function. We discuss each of these pieces in turn for our example.

As noted, the WorldNet session data is stored in an ASCII format. Each line contains a record. Each record has four fields, separated by white space. A typical entry has the form:

```
100198122 atlgalocal       925747790        106
```

The first field stores the WorldNet customer's identification number. The second field is a string that stores the name of the modem bank that the customer called. The third field encodes the time at which the connection was made, including both the date and the time of day. The fourth field denotes the duration of the session.

For each field in the physical representation, only certain values are legal. For example, customer identification numbers fall into a particular range. Any value outside that range is not a legal identification number. Understanding what constitutes a legal record is an important part of understanding the physical representation of the data.

We can represent the logical record for WorldNet sessions with the following C `struct`:

```
#define NAME_LENGTH 30

/* Logical representation of WorldNet session.*/
/* pop names have short bounded length,
 * so use fixed-width representation. */
typedef struct {
  userID_t id;
  char pop[NAME_LENGTH + 1];
  Hdate_t connect_date;
  Htime_t connect_time;
  int duration;
} session_t;
```

In `session_t`, we have chosen to limit the length of the name of the modem bank to `30` characters. We could have used the hash-table pickle type (`pht_t`) defined in Section 8.1 as an alternative to fixed length strings.

The translation function for the WorldNet session data source is shown in Figure 9.5. This function takes a `Sfio_t` pointer[3] as its first argument and space to store a logical record as its second argument. It returns an integer flag to signal the success of the translation.

The body of the function uses `sfscanf` to read a line of input, which corresponds to a physical record for this stream. If `sfscanf` reaches the end of the file before reading the necessary four values, the function returns the `HRS_STREAM_DROP_REC` to signal that it could not find a valid record. It does not return `HRS_STREAM_STOP` because there may be another file associated with the stream that contains more records. If `sfscanf` found four values, the code ensures that the length of the string storing the name of the modem center will fit in the logical record and that the `id` number fits within its logical bounds. If these conditions are met, the function copies the data into `s` and returns `HRS_STREAM_KEEP_REC` to signal success.

[3] `Sfio` provides equivalents to all the standard I/O routines, e.g., printf, scanf, etc.

```
int getValidSession(Sfio_t *input, session_t *s)
{
  userID_t id;
  char tpop[1000];
  int utime;
  int duration;
  int found;

  if ((found = sfscanf(input, "%d %s %d %d\n",
                 &id, &tpop, &utime, &duration)) == EOF)
    return HRS_STREAM_DROP_REC;

  if (found != 4)
    return HRS_STREAM_DROP_REC;

  if (strlen(tpop) > NAME_LENGTH)
    return HRS_STREAM_DROP_REC;
  else
    strcpy(s->pop, tpop);

  s->connect_date = extractDATE(utime);
  s->connect_time = extractTIME(utime);

  s->duration = duration;

  if ((id < MIN_ID) || (id > MAX_ID))
    return HRS_STREAM_DROP_REC;
  s->id = id;

  return HRS_STREAM_KEEP_REC;
}
```

Figure 9.5: Stream translation function for WorldNet records.

In general, the translation function for a general stream should return `HRS_STREAM_DROP_REC` if given an empty line of input, because the function may be called an extra time to detect the end of each file in the stream.

Once the stream is defined a programmer can view a general stream as a stream of logical records; details about the representation of the data on disk are confined to the translation function.

```
int convert(char set, pht_p pht, hash_iter i, unsigned int *h){
  int result;
  char *s;

  if (!set) return HRS_STREAM_ERROR;
  result = next_iter_pht(pht, i, &s, h);
  if (PHT_ERROR == result)
    return HRS_STREAM_ERROR;
  else if (HASH_ITER_FOUND == result)
    return HRS_STREAM_KEEP_REC;
  else {
    close_iter_pht(pht, i);
    i = init_iter_pht(pht);
    return HRS_STREAM_STOP;
  }
}
```

Figure 9.6: Translation function for a generative stream that converts the keys of a persistent hash table into a stream.

9.5.4 Example: Persistent hash table stream

In this section, we define a generative stream:

```
stream pht_s (pht_p p, hash_iter i ) {
        convert(:p, i:) : void => unsigned int;
};
```

that converts a persistent hash table into a stream of hash codes with values stored in the table. The logical representation for this stream is the type of the hash code: an `unsigned int`. The translation function, `convert`, which appears in Figure 9.6, takes a persistent hash table and an associated iterator as arguments. Because `convert` depends critically on the type parameter, it returns `HRS_STREAM_ERROR` to halt the program if `set` is false. Otherwise, it gets the next key from the hash table using the argument iterator and checks the return code. It uses `HRS_STREAM_ERROR` to report an error or `HRS_STREAM_KEEP_REC` to signal that it found the next key. Otherwise, the iterator has reached the end of the hash table. In this case, we close the old iterator and initialize a new one, so successive iterations over the stream will start fresh. Finally, we return `HRS_STREAM_STOP` to terminate the stream. Translation functions for generative streams that do not return eventually either `HRS_STREAM_ERROR` or `HRS_STREAM_STOP` will cause an infinite loop in iteration.

Generative streams do not have their own on-disk representations. Because of this fact, we use the constant `NULL` as the path expression in initializing declarations for generative streams.

The sample code in Figure 9.7 illustrates using a generative stream to print all the strings stored in a persistent hash table by iterating over the associated generative stream of hash codes.

```
pht_s createPHTstream ( pht_p p ) {
  hash_iter i = init_iter_pht(p);
  pht_s s(:p,i:) = NULL;
  return s;
}

void sig_main(pht_p p <p:, "Persistent hash table">){
  pht_s ps = createPHTstream(p);
  iterate(over ps){
    event (unsigned int *h){
      sfprintf(sfstdout, "%s\n", lookup_pht(p, *h));
    }
  }
}
```

Figure 9.7: Translation function for a generative stream that converts the keys of a persistent hash table into a stream.

This code wraps the initializing declaration in the function `createPHTstream` so it can initialize the iterator. It then uses the internal event detection function associated with `pht_s` to consume the hash keys. The event code looks up the hash key in the persistent hash table and prints the associated string.

Chapter 10

Sig_main

Hancock provides the `sig_main` construct to connect command-line input to variables in Hancock programs. As an example, consider the following `sig_main` function from the Usage signature:

```
int sig_main(callDetail_s calls <c:>,
    exists const usage_m oldUsage <u:, "Yesterday's map">,
    new          usage_m newUsage <U:, "Today's map">)
{
  /* arbitrary Hancock code */
}
```

The `calls` parameter is a stream that contains call-detail data. The syntax (<c:>) after the variable name specifies that this parameter will be supplied as a command-line option using the `-c` flag. The colon indicates that the flag takes an argument, in this case the name of the directory that holds the call files. The absence of a colon indicates that the parameter is a boolean flag. The `oldUsage` identifier is a `usage_m` map, the name of which is specified using the `-u` flag. The `const` qualifier indicates the map is read-only, while the `exists` annotation requires the map to exist on disk. The optional string `"Yesterday's map"` documents the intended semantics for the parameter. The `newUsage` parameter is another `usage_m` map. The `-U` flag specifies the file name for this map, and the `new` qualifier indicates that the map must not previously exist on disk.

If a `sig_main` function appears in a Hancock program, it assumes the role of the `main` function in standard C programs. It is an error to have more than one `sig_main` function or to have both a `sig_main` and a `main` function. The return type of a `sig_main` function may be either `void` or `int` to return a status code.

If a program contains a `sig_main` declaration, the Hancock compiler generates command-line processing code to connect command-line inputs to `sig_main` parameters. In addition, the compiler generates code to support the `--` command-line option. When a program is invoked with this switch, the program returns on `stdout` a string describing the switches available from the program, annotated with any user-supplied documentation. For example, if the above program were called `usage.exe`, then entering `"usage.exe --"` at the command line would yield:

```
usage.exe: usage:
        -c:
        -u: (Yesterday's map)
        -U: (Today's map)
```

10.1 Supported types

Hancock supports map, pickle, directory, and stream types as `sig_main` parameters, as well as all of the basic C types, except for `long doubles`. Hancock also supports the portable types defined in Table 12.1 on page 66. The C type `char *` is supported, in which case the command-line argument is interpreted as a string.

The following code illustrates using persistent Hancock types on the command-line:

```
void sig_main( qualifiers aMap_m    myMap  <m:>,
               qualifiers aPickle_p myPic  <p:>,
               qualifiers aDir_d    myDir  <d:>,
               qualifiers aStream_s myStr  <s:, "Today's data!">)
  {
      ...
  }
```

In this case, these declarations serve as a special form of initializing declaration (Chapter 5), in which the string supplied at the command-line is interpreted as the path name to the on-disk representation of the corresponding Hancock value. The optional qualifier lists play the same role as they do in initializing declarations.

The following `sig_main` function illustrates using basic C types in addition to Hancock types as `sig_main` parameters:

```
void sig_main(int i <i:>,
              char c <c:>,
              double d <d:>,
              char * s <s:>,
              usage_m m <u:, "Usage map">)
{
      ...
}
```

If the program containing this function were called `foo`, then the invocation

```
foo -i 12 -c 'c' -d 0.85 -s "Hello, world!" -m data/usageMap
```

would cause the variables `i`, `c`, `d`, and `s` to be initialized to the values `12`, `'c'`, `0.85`, and `"Hello, world!"`, respectively. The map `m` would be associated with the on-disk map stored in the file `data/usageMap`. At run-time, the system checks that the value supplied at the command-line can be converted to a value of the associated type. If it cannot, a run-time error is generated and the program halts.

10.2 Defaults

Hancock provides a *default* mechanism to specify values to associate with `sig_main` parameters that are omitted from the command-line. Each `sig_main` parameter may

contain a `default` clause, as in:

```
void sig_main(int i <i:> default 0,
              char c <c:> default 'a',
              double d <d:> default 0.0,
              char * s <s:> default "Default string!",
              usage_m m <u:, "Usage map"> default "data/oldMap")
{
      ...
}
```

Each default expression must be a constant expression with a type that matches the type declared for the associated parameter. Parameters that have Hancock types can be initialized with either values of the same type or of type `char *`. In the latter case, the string is interpreted as the path to the on-disk representation of the data. If the program `default-foo` contained this version of `sig_main`, then the invocation

```
default-foo  -d 0.85 -s "Hello, world!"
```

would cause the variables `i` and `c` to have values `0` and `'a'`, respectively. Variables `d` and `s` would get values `0.85` and `"Hello, world!"` as before. Map `m` would be associated with the data in file `data/oldMap`. If no value is supplied at the command line for a `sig_main` parameter without a default, the system raises a run-time error.

10.3 Boolean flags

In addition to value-carrying flags, Hancock supports boolean flags. Intuitively, the associated Hancock variable is true if the flag is present on the command line and false otherwise. The following `sig_main` declaration specifies such a flag:

```
void sig_main(char bv <b>) {
      ...
}
```

The lack of a colon after the switch name `b` indicates that this flag is conceptually a boolean, not a `char`, since the switch does not expect an argument. The type associated with boolean flags must be a `char`.

10.4 Standard input

It is convenient occasionally to initialize a `sig_main` stream parameter from standard input to the command line, *e.g.*, when a programmer wishes to redirect the output of another program to a Hancock program. Hancock supports this idiom using either `stdin` or `sfstdin` as a special command-line argument

Any stream parameter passed `stdin` or `sfstdin` at the command-line takes its data from standard input. For example, if `sig_main` for a program `foo` is

```
void sig_main(callDetail_s myStream <s:>) {
      ...
}
```

then the invocation

```
generateData | foo -s stdin
```

causes the output for `generateData` to be the data associated with `myStream` in program `foo`.

The special strings `stdin` and `sfstdin` may also be used in the default clause for stream parameters, to the same effect. For example, if program `foo` contains the following `sig_main` declaration:

```
void sig_main(callDetail_s myStream <s:> default "sfstdin") {
      ...
}
```

then variable `myStream` will be connected to standard input if the `s` switch is omitted from the command-line, as would be the case in the invocation:

```
generateData | foo
```

10.5 Program name

It is often convenient for a program to know its name, for example, in reporting error messages. To support this idiom, Hancock sets the value of the first `sig_main` parameter to the name of the containing program if that parameter has type `char *` and it specifies no command-line switch. For example, if the following program is called `getI`:

```
#include <stdio.h>

int sig_main(char * program_name,
             int i <i:, "Command-line integer">)
{
   printf("%s: Switch i had value: %d.\n", program_name, i);
}
```

then invoking `getI -i 10` will print:

```
getI: Switch i had value: 10.
```

Chapter 11

Hancock library

This chapter describes the library that is distributed with Hancock. At present, the library provides only a few stream data types and a few functions, but we expect it to grow over time. Programs that use the type declarations and functions in the library must include the header file `hl.hh`.

The Hancock library supports a general stream type (`HLkey_s`) for streams of keys (see Figure 11.1). The stream type takes two parameters that together define the (inclusive) range of valid keys for the stream. The keys are represented in ASCII, one key per line, in the physical representation. The logical representation is a `long long`. This stream is very useful for writing programs that take a map and a list of keys and select data for the keys from the map. See Figure 11.2 for an example use of this stream.

The Hancock library also supports a stream type and operations for converting maps into streams; the associated declarations appear in Figure 11.3. The stream type, `HLMapStream_s`, comes with three functions for creating streams of that type: `HLcreateMapStream`, `HLcreateMapMerge2`, and `HLcreateMapMerge3`.

The first function, `HLcreateMapStream`, takes a map (`m`) along with the desired range of keys. The parameter `all` is a boolean. `True` means that the programmer wants the stream to have all active keys from the map in the stream, whereas `false` means that programmer wants only keys from the (inclusive) range specified by the `low` and `high` parameters. This stream provides similar functionality to built-in map-to-stream expression, but is more general. It can be used in any context that allows a

```
int HLgetValidKey(char set, long long low, long long high,
                  Sfio_t *input, long long *pn);

stream HLkey_s(long long low, long long high)
       { HLgetValidKey(: low, high :) : Sfio_t => long long; };
```

Figure 11.1: `HLkey_s` stream definition.

```
#include "hl.hh"
void sig_main(exists intlCount_m ic <I:, "intlCount map">)
{
  HLkey_s lines(:MINVALIDPN, MAXVALIDPN-1:) = "sfstdin";

  iterate
   ( over lines ) {

   event (long long *pn) {
      sfprintf(sfstdout,"%lld ==> %d\n", *pn, ic<:*pn:>);
   }
  };

}
```

Figure 11.2: Example use of `HLkey_s`.

```
stream HLmapStream_s(int (*fn)(char set, void *v, long long *pn),
                     void *v)
       { fn(:v:) : void => long long; };


HLmapStream_s HLcreateMapStream(HRSmap_t m, char all,
                                long long low, long long high);

#define OR_MERGE 0
#define AND_MERGE 1

HLmapStream_s HLcreateMapMerge2(HRSmap_t m1, HRSmap_t m2, char all,
                                long long low, long long high,
                                char operation);

HLmapStream_s HLcreateMapMerge3(HRSmap_t m1, HRSmap_t m2, HRSmap_t m3,
                                char all, long long low, long long high,
                                char operation);
```

Figure 11.3: `HLmapStream_s` stream definition.

stream, not just in an iteration statement.

The second function, `HLcreateMapMerge2`, produces a stream that merges the active keys from two maps. The function takes two map arguments, a range of keys using the same arguments as `HLcreateMapStream`, and an operation. At this time, the only operations that are supported are `OR_MERGE` and `AND_MERGE`. The first produces a stream of keys where each key in the stream is active in at least one map. The second produces a stream of keys where each key in the stream is active in both maps.

The third function, `HLcreateMapMerge3`, provides the same functionality as the second, but for three maps rather than two. The merge-map streams are useful for doing computations over several maps that share overlapping ranges of keys, but may not have identical sets of active keys.

Chapter 12

Portability issues

Different computer architectures represent data differently. Both the number of bytes and the byte order used for an `int`, for example, vary from machine to machine. The format used on one machine may not be understood on another machine. To write portable Hancock programs, there are three issues to consider:

1. To avoid overflow, the exact number of bytes needed to hold each variable must be specified.
2. Data to be transferred between platforms must be converted from the native format on the first platform to an agreed upon, portable format. Later, when the data is read on a second platform, it must be converted from the portable format to the native format of the second platform.
3. System-specific function calls, constant declarations, and format characters must be replaced with agreed upon macro definitions.

Hancock includes a collection of C types, functions, and macros to facilitate writing portable code.

12.1 Portable types

Portable types address the first issue. Such types are guaranteed to use the same number of bytes on all platforms. To understand why a fixed-size is important, consider a program written on a computer where a `long` is 64 bits and an `int` is 32 bits. Suppose further that the programmer declared a variable `v` with type `long` to store a value too large to fit in 32 bits. If this program were run on a machine where a `long` is defined to be 32 bits, there will be an overflow during the execution of the program, which most likely will result in corrupted data. If the programmer instead had used the portable type `int64`, which is guaranteed to be 64 bits on all platforms, this problem would have been avoided. Table 12.1 lists the portable types provided in Hancock and their corresponding sizes.

Table 12.1: Portable types and their corresponding sizes.

Type	Number of bits
signed integers	# of bits
`int8`	8
`int16`	16
`int32`	32
`int64`	64
unsigned integers	# of bits
`uint8`	8
`uint16`	16
`uint32`	32
`uint64`	64
floating point values	# of bits
`float32`	32
`float64`	64

12.2 Portable format

The crux of the second issue is byte order, which varies from machine to machine. To write a value larger than a byte to a file in a portable way, the order of the bytes must be converted to an agreed-upon, portable format that can be understood on all machines.

For Hancock data to be portable, the persistent representations of directories, streams, and maps must be in this portable format. Hancock directories store their persistent data in ASCII, which is byte-based and therefore portable. Currently, Hancock does not generate streams, so the portability of stream data relies on the data source. Hancock maps contain two kinds of data: internal indexing data and compressed user data. The first form of data is always written in a portable format. The compressed user data is in portable form if the map compression function fills its byte buffer with bytes in a portable form. In this case, the decompression function reads the bytes out of its buffer and translates them to native form.

For maps without user-supplied compression functions, Hancock generates compression functions automatically. The default compression routines generate portable data if all the base types in the map value type are portable types, as described in Section 12.1. Otherwise, the default compression routines are not portable.

Maps with user-supplied compression are portable if the supplied compression routines read and write data in a portable format. To facilitate writing portable compression functions, Hancock provides conversion functions that translate between the native data format on the current machine and a portable format called the *network format*. Table 12.2 lists the API's of these functions.

Table 12.2: Translation functions between the data format on the host and the portable data format. The portable format is also known as the network format. In the `hton_X()`, `ntoh_X()`, `store_X()`, and `set_X()` functions, `a` is the value to be converted. In the `read_X()`, and `get_X()` functions, `a` is a pointer to the value to be converted. In the `store_X()`, `read_X()`, `set_X()`, and `get_X()` functions, the value is read from, or written to, a buffer. The `store_X()` and `read_X()` functions increment the pointer to the buffer, whereas the `set_X()` and `get_X()` functions do not. It is the responsibility of the caller to allocate sufficient memory for the buffer.

From host to network format	From network to host format
Signed integers	Signed integers

`int8 hton_int8(int8 a);`
`int16 hton_int16(int16 a);`
`int32 hton_int32(int32 a);`
`int64 hton_int64(int64 a);`
	`int8 ntoh_int8(int8 a);`
	`int16 ntoh_int16(int16 a);`
	`int32 ntoh_int32(int32 a);`
	`int64 ntoh_int64(int64 a);`

`void store_int8(int8** b, int8 a);`
`void store_int16(int8** b, int16 a);`
`void store_int32(int8** b, int32 a);`
`void store_int64(int8** b, int64 a);`
	`void read_int8(int8** b, int8* a);`
	`void read_int16(int8** b, int16* a);`
	`void read_int32(int8** b, int32* a);`
	`void read_int64(int8** b, int64* a);`

`void set_int8(int8* b, int8 a);`
`void set_int16(int8* b, int16 a);`
`void set_int32(int8* b, int32 a);`
`void set_int64(int8* b, int64 a);`
	`void get_int8(int8* b, int8* a);`
	`void get_int16(int8* b, int16* a);`
	`void get_int32(int8* b, int32* a);`
	`void get_int64(int8* b, int64* a);`

Unsigned integers	Unsigned integers

`uint8 hton_uint8(uint8 a);`
`uint16 hton_uint16(uint16 a);`
`uint32 hton_uint32(uint32 a);`
`uint64 hton_uint64(uint64 a);`
	`uint8 ntoh_uint8(uint8 a);`
	`uint16 ntoh_uint16(uint16 a);`
	`uint32 ntoh_uint32(uint32 a);`
	`uint64 ntoh_uint64(uint64 a);`

`void store_uint8(int8** b, uint8 a);`
`void store_uint16(int8** b, uint16 a);`
`void store_uint32(int8** b, uint32 a);`
`void store_uint64(int8** b, uint64 a);`
	`void read_uint8(int8** b, uint8* a);`
	`void read_uint16(int8** b, uint16* a);`
	`void read_uint32(int8** b, uint32* a);`
	`void read_uint64(int8** b, uint64* a);`

`void set_uint8(int8* b, uint8 a);`
`void set_uint16(int8* b, uint16 a);`
`void set_uint32(int8* b, uint32 a);`
`void set_uint64(int8* b, uint64 a);`
	`void get_uint8(int8* b, uint8* a);`
	`void get_uint16(int8* b, uint16* a);`
	`void get_uint32(int8* b, uint32* a);`
	`void get_uint64(int8* b, uint64* a);`

Floating point values	Floating point values
`float32 hton_float32(float32 a);` `float64 hton_float64(float64 a);`	`float32 ntoh_float32(float32 a);` `float64 ntoh_float64(float64 a);`
`void store_float32(int8** b, float32 a);` `void store_float64(int8** b, float64 a);`	`void read_float32 (int8** b, float32* a);` `void read_float64 (int8** b, float64* a);`
`void set_float32(int8* b, float32 a);` `void set_float64(int8* b, float64 a);`	`void get_float32(int8* b, float32* a);` `void get_float64(int8* b, float64* a);`

Table 12.3: Portable definitions.

Endian	
LITTLE_ENDIAN	0 or 1; 1 iff it is a little endian machine
BIG_ENDIAN	0 or 1; 1 iff it is a big endian machine
MIDDLE_ENDIAN	0 or 1; 1 iff it is neither a big nor little endian machine
Compiler specific definitions	
MIN_ARRAY_LENGTH	0 or 1; minimum number of entries in an array
VAR_ARRAY_LENGTH	0 or 1; 1 iff compiler supports variable length arrays
TYPE_OF	0 or 1; 1 iff compiler supports typeof expressions
ALIGN_DATA	0 or 1; 1 iff computer needs data to be aligned
CHAR_SIGNED	0 or 1; 1 iff char is signed
C/C++ interoperability	
__BEGIN_DECLS	begin of extern "C" declarations
__END_DECLS	end of extern "C" declarations
intX constants	
I**X**_C(v)	signed integer constant for `int`**X**, *e.g.*, I64(14) is a 64bit integer of value 14
UI**X**_C(v)	unsigned integer constant for `uint`**X**, *e.g.*, UI64(14) is a 64bit unsigned integer of value 14
F**X**_C(v)	float constant for `float`**X**, *e.g.*, F64(23.3) is a 64bit float of value 23.3
`printf` labels	
D**X**_L	"%d" for `int`**X**, *e.g.*, `printf(``I='' D64_L ``\n'', (int64) I);`
U**X**_L	"%u" for `uint`**X**
X**X**_L	"%x" for `int`**X**
E**X**_L	"%e" for `float`**X**
G**X**_L	"%g" for `float`**X**
F**X**_L	"%f" for `float`**X**
OS specific definitions	
IS_LINUX	0 or 1; 1 iff the operating system is Linux
IS_IRIX	0 or 1; 1 iff the operating system is IRIX
IS_SOLARIS	0 or 1; 1 iff the operating system is SOLARIS
clib specific definitions	
HAS_SNPRINTF	0 or 1; 1 iff snprintf is in clib
HAS_VSNPRINTF	0 or 1; 1 iff vsnprint is in clib
RLIMIT_VMEM	defined if <sys/resource.h> does not define
RTLD_NEXT	defined if <dlsym.h> does not defined

12.3 System-specific definitions

The third issue involves platform-dependent definitions. Hancock provides a set of macro definitions to enable the use of such definitions. This set is the same on all machines, but each macro might evaluate differently on different machines. For example, the macro `IS_LINUX` is defined an all platforms but evaluates to `0` on all platforms other than Linux, where it is `1`. Another example is the `printf` function. When giving the formatting string, the format characters depend on the type being written, for example `%d` for `int`s and `%ld` for `long`s. But when using portable types, only the size of the type is known, not its name. Table 12.2 shows the special format characters for printing portable types and the other portability macros that Hancock defines.

Chapter 13

Extended example: The Cell Tower Application

In this chapter, we illustrate the features of the Hancock language by using them to implement an application that mines information from a wireless call-detail stream. In particular, the application, called the Cell Tower application, measures the mobility or "diameter" of mobile phone numbers (MPNs). Phone numbers that are used exclusively in one or a few neighboring cells have small diameters, while those used in larger regions have larger diameters. Such information is useful for fraud detection and for developing new location-based services.

The wireless call-detail stream consists of a sequence of records, each one of which describes a call made on the wireless network. Although these records contain many fields, only the following few are relevant for our purposes:

- Originating phone number
- Dialed phone number
- Primary cell tower (originating)
- Secondary cell tower (originating)
- Primary cell tower (dialed)
- Secondary cell tower (dialed)

Either one (or both) of the originating and dialed phone numbers correspond to a mobile phone number (MPN). Information related to mobility appears in the cell tower fields. If the originating phone number belongs to a mobile phone, then the originating primary tower captures the first tower used to carry the call. If the phone moved significantly, then the secondary tower captures the last tower used during the call. The dialed towers carry the same information for the dialed phone number.

The Cell Tower application tracks for each MPN the five most frequently (and most recently) used cell towers and another value that captures the frequency with which calls placed to/from the MPN do not involve the top five cell towers. As one might expect, the top five list is dynamic, so the signature computation includes a

```
#define TOPN 5
#define CTD PHT_UNKNOWN

typedef struct {
  unsigned int tower[TOPN];
  float count[TOPN];
  float other;
} profile_t;
```

Figure 13.1: C-type declaration describing mobile phone number profile.

probabilistic bumping algorithm that allows a new cell tower to enter the top five list as its frequency of use increases.

More concretely, we will use the C struct, `profile_t`, shown in Figure 13.1 as the type of the signature for each mobile phone number. The `tower` array stores the five most frequently used cell towers, while the parallel array `count` measures the frequency with which the corresponding tower is used. Field `other` measures how many calls are not reflected in the list of the top five towers.

The rest of this chapter discuss the persistent data types that we use to represent the Cell Tower signature, the stream data type that we use to represent the stream of wireless call detail records, and finally the computation used to update the Cell Tower signatures from the call stream.

13.1 Cell Tower: persistent data

In the Cell Tower application, we want to track phone number profiles over time; consequently, we use the Hancock map shown in Figure 13.2 to associate each mobile phone number with its profile persistently. We review this definition briefly. The key clause from the declaration indicates mobile phone numbers are represented by C values of type `long long` that fall between `MINPN` and `MAXPN`, inclusively. The `split` clause says that I/O to transfer between the on-disk and the in-memory representations will be done in units of 10,000 values and compression will be done in units of 100 values. The `default` clause specifies that the map should return the constant `CTM_DEFAULT` when asked for the value associated with a key that has no value stored in the map. This value gives the constant `CTD` as the default value for cell-tower hash values and zeros for the counts. The compression/decompression clauses specify functions for compressing and decompressing single `profile_t` values. These functions, which appear in Figure 13.3, use a table, passed in as an argument (`ctab`) to the map type, for compression and decompression. In this example application, the compression table is used only to show how such tables might be passed to a map. The compression table pickle type declaration `ct_p` appears in Figure 13.4; the associated functions are shown in Figure 13.5.This pickle takes one parameter, `level`, intended to represent the desired compression level, with higher values indicating a greater willingness to spend time to produce a better compression factor. As with the compression functions, this code merely sketches the structure of a compression table pickle.

```
#include "ctp.hh"
int ctSqueeze(char set, ct_p ctab, profile_t *from,
              uint8 *buffer, int32 bufferSize);

int ctUnsqueeze(char set, ct_p ctab, uint8 *from,
                int32 bufferSize, profile_t *to);

#define CTM_DEFAULT  {{CTD, CTD, CTD, CTD, CTD}, \
                      {0.0, 0.0, 0.0, 0.0, 0.0}, \
                      0.0}

map cellTower_m(ct_p ctab)  {
  key (MINPN .. MAXPN);
  split (10000, 100);
  value profile_t;
  default CTM_DEFAULT;
  compress ctSqueeze(:ctab:);
  decompress ctUnsqueeze(:ctab:);
};
```

Figure 13.2: Map declaration for the Cell Tower application.

In the `profile_t struct`, we use an unsigned integer to represent cell towers. In the wireless call-detail stream, however, cell towers are represented as variable-length strings. Because types with fixed size are much more convenient for both computation and storage, we use a hash table to associate an (unsigned) integer hash key with each string and store the hash key in the profile instead of the string. Because the profile data is persistent, the mapping between a given hash key and a given cell tower name must be persistent as well.

The Cell Tower application uses `pht_p`, a Hancock pickle type, to define a persistent hash table. The function prototypes and type declarations for this pickle appear in Figure 13.6. The in-memory representation for pickles of this type is a C struct, `pht_rep`, that contains an in-memory hash table, an indication of whether the structure is readonly, and an indication whether the hash table has been updated since it was opened. The function `init_pht` is used to initialize pickles of this type and the function `flush_pht` is used to write the hash table back to disk on exit. This data type also provides a collection of functions for manipulating the hash data: `insert_pht` adds a new item into the hash table, if necessary; `lookup_pht` returns the string associated with a hash value; and `init_iter_pht`, `next_iter_pht`, and `close_iter_pht` implement an iterator for the hash table. The code for these functions is shown in Figure 13.7 and Figure 13.8.

We use a Hancock directory, `cellTower_d`, to reflect the close semantic coupling in the application program among the signature collection, the persistent hash table, and the compression table to ensure that we use the persistent data properly. The declaration for `cellTower_d`, shown in Figure 13.9, takes one parameter, `level`, which it passes to the compression table pickle as a parameter to control the degree of

```
int ctSqueeze(char set, ct_p ctab, profile_t *from,
              unsigned char *buffer, int32 bufferSize){
  /* Halt if no compression table. */
  if (!set) return HRS_ERROR;

  if (0 == ctab->level % 2)
    from->other =  -1 * from->other;
  if (sizeof(profile_t) > bufferSize)
    return HRS_ERROR;
  memcpy ((profile_t *)buffer,
          (void const *)from, sizeof(profile_t));
  return sizeof(profile_t);
}

int ctUnsqueeze(char set, ct_p ctab, uint8 *buffer,
                int32 bufferSize, profile_t *to){
  /* Halt if no compression table. */
  if (!set) return HRS_ERROR;

  if (sizeof(profile_t) > bufferSize)
    return HRS_ERROR;
  memcpy (to, (void const *)buffer, sizeof(profile_t));
  if (0 == ctab->level % 2)
    to->other = -1 * to->other;
  return sizeof(profile_t);
}
```

Figure 13.3: User-supplied compression functions for the Cell Tower map.

```
typedef struct {
  int level;
  char readonly;
} ct_rep;

int init_ct(char set, int level,
            Sfio_t *fp, ct_rep *data, char readonly);
int flush_ct(Sfio_t *fp, ct_rep *data, char close);

pickle ct_p(int level) {init_ct(:level:) => ct_rep => flush_ct};
```

Figure 13.4: Template compression table pickle for use with Cell Tower map.

```
#define COMPRESSION_LEVEL_DEFAULT   6

/* truncate file fd to length. */
int ftruncate (int fd, off_t length);

int init_ct(char set, int level, Sfio_t *fp,
            ct_rep *data, char readonly){
  data->readonly = readonly;
  if (0 == sfsize(fp)) {     /* empty file */
    if (set)                        /* Was parameter given? */
      data->level = level;          /* Yes: Use parameter */
    else                            /* No:  Use default */
      data->level = COMPRESSION_LEVEL_DEFAULT;
    return HRS_OK;
  } else {
    int result = sfscanf(fp, "%d", &data->level);
    if (1 == result)
      return HRS_OK; /* Retrieved compression level from disk. */
  }
  return HRS_ERROR; /* Couldn't initialize compression table. */
}

int flush_ct(Sfio_t *fp, ct_rep *data, char close){
  Sfoff_t curLoc;

  /* Seek to beginning of the file. */
  sfseek(fp, (Sfoff_t) 0, SEEK_SET);
  sfprintf(fp, "%d\n", data->level);

  /* Find current seek address. */
  curLoc = sfseek(fp, (Sfoff_t) 0, SEEK_CUR);

  /* Truncate file fp to curLoc (end of currnet contents).  */
  ftruncate(sffileno(fp), curLoc);
  return HRS_OK;
}
```

Figure 13.5: Initialization and flushing functions for the template compression table pickle.

```
#include "hash.h"
#define PHT_ERROR -1
#define PHT_UNKNOWN HASH_DEFAULT
typedef struct {
  hash_table *ht;
  char readonly;
  char updated;
} pht_rep;

int init_pht(Sfio_t *fp, pht_rep *data, char readonly);

int flush_pht(Sfio_t *fp, pht_rep *data, char close);

pickle pht_p {init_pht => pht_rep => flush_pht};

int insert_pht(pht_p data, char *s, unsigned int *h);
char *lookup_pht(pht_p data, unsigned int h);

hash_iter init_iter_pht(pht_p data);
int next_iter_pht(pht_p data, hash_iter i,
                  char **s, unsigned int *h);
void close_iter_pht(pht_p data, hash_iter i);
```

Figure 13.6: Function prototypes and type declarations for the `pht_p` pickle type.

compression.

13.2 Wireless call detail stream

The Cell Tower application uses the general stream description shown in Figure 13.10. The translation function `getValidCall` takes a persistent hash table for translating cell tower strings into hash values as a parameter. The function extracts the next valid record from a file, checks the validity of the mobile phone numbers, and translates the cell tower strings into hash values. The code for this function is shown in Figure 13.11.

13.3 Control flow

The Cell Tower application uses a process flow typical for signature applications to compute the desired persistent information. Figure 13.12 depicts this flow: transaction records are collected for some time period, the length of which depends on the application (*e.g.*, a day for marketing but just a few minutes for fraud detection). At the end of the time period, the records are processed to update the signatures. Before processing, the old signature data is copied to preserve a back-up for error-recovery purposes and the existing data is aged. Finally, the outgoing map is updated from the call stream and the incoming map is updated from the call stream.

```c
/* File representation:
 * hash0:len:string_0
 * hash1:len:string_1
 * ...
 * hashn:len:string_n
 */

int init_pht(Sfio_t *fp, pht_rep *data, char readonly){
  int result, len;
  unsigned int hash;
  char *s;
  data->ht = hash_empty();
  data->readonly = readonly;
  data->updated = 0;
  if (0 == sfsize(fp)) return HRS_OK; /* empty file, okay */
  else
    while (!sfeof(fp)) {
      if (2 != sfscanf(fp, "%u:%d:", &hash, &len))
        return HRS_ERROR;
      s = (char *)malloc(len+1);
      if (NULL == s) return HRS_ERROR;
      if (1 != sfscanf(fp, "%s\n", s)) return HRS_ERROR;
      if (HASH_OK != hash_insert_pair(data->ht, s, hash))
        return HRS_ERROR;
    }
  return HRS_OK; /* okay */
}

int flush_pht(Sfio_t *fp, pht_rep *data, char close){
  /* add error checking. */
  int len;
  char *s;
  unsigned int hash;
  hash_iter i;

  if ((!data->readonly) && (data->updated)) {
    Sfoff_t curLoc;
    sfseek(fp, (Sfoff_t) 0, SEEK_SET);
    i = init_iter_pht(data);
    while (HASH_ITER_FOUND == next_iter_pht(data, i, &s, &hash)) {
      sfprintf(fp, "%u:%d:", hash, strlen(s));
      sfprintf(fp, "%s\n", s);
    }
    close_iter_pht(data, i);
    curLoc = sfseek(fp, (Sfoff_t) 0, SEEK_CUR);
    ftruncate(sffileno(fp), curLoc);
  }
  if (close)
    hash_close(data->ht);
  return HRS_OK; /* okay */
}
```

Figure 13.7: Initialization and flushing functions for the `pht_p` pickle.

```
int insert_pht(pht_p data, char *s, unsigned int *h){
  if (!data->readonly){
    data->updated = 1;
    return hash_insert(data->ht, s, h);
  } else
    return PHT_ERROR;
}

char* lookup_pht(pht_p data, unsigned int h){
  return hash_lookup(data->ht, h);
}

hash_iter init_iter_pht(pht_p data){
  return hash_init_iter(data->ht);
}

int next_iter_pht(pht_p data, hash_iter i,
                  char **s, unsigned int *h){
  return hash_next(data->ht, i, s, h);
}

void close_iter_pht(pht_p data, hash_iter i){
  hash_close_iter(data->ht, i);
}
```

Figure 13.8: Implementations of auxiliary functions for the `pht_p` pickle.

```
directory cellTower_d(int level) {
  pht_p ctHashTable;
  ct_p  ctab(:level:);
  cellTower_m outMap(:ctab:);
  cellTower_m inMap(:ctab:);
  char *lastUpdated default "never";
}
```

Figure 13.9: Directory declaration for Cell Tower application.

```
#include "pht.hh"

typedef struct {
   long long origin;
   long long dialed;
   unsigned int poct;       /* PHT_UNKNOWN if not in data. */
   unsigned int soct;       /* PHT_UNKNOWN if not in data. */
   unsigned int pdct;       /* PHT_UNKNOWN if not in data. */
   unsigned int sdct;       /* PHT_UNKNOWN if not in data. */
} wc_t;

int getValidCall(char paramSet, pht_p p,
                 Sfio_t *file, wc_t* lcr);

stream wireless_s(pht_p p)
           { getValidCall(: p :) : Sfio_t => wc_t; };
```

Figure 13.10: General stream declaration for the Cell Tower application.

Figure 13.13 shows the code for aging the data. The function `age` takes a `ctMap` and iterates over it. For each active key, the function retrieves the associated values, ages them, and then writes them back into the map. This function is called once for each `ctMap`.

Figure 13.14 shows the code for updating the outgoing Cell Tower map. This function, `doOrigin`, iterates over the wireless call stream. First, the calls are filtered using the user-defined function `completeOriginCellCall` to remove incomplete calls and calls that did not originate from a mobile phone. Next, the records are sorted according to the originating phone number. This function uses three events— `line_begin`, `call`, and `line_end`— that are detected using the function `originDetect`. The multi-union that defines these events appears in Figure 13.15, while the function `originDetect` appears in Figure 13.16.

For each line seen in the stream, the computation computes an approximation to "today's" top five cell towers by starting with an empty profile and merging in each cell-tower as it is seen in a call. Once all the calls for the line have been seen, the line's signature is retrieved from disk, this historical data is integrated with "today's" data, and the resulting signature is written back to disk.

Figure 13.17 shows the code for updating the incoming Cell Tower map. The computation is very similar. The calls are filtered to remove incomplete calls and calls that did not terminate at a mobile phone. The remaining calls are sorted by the dialed phone number. Function `dialedDetect` triggers events based on the dialed phone number in the record. Finally, an approximation of "today's" profile is computed, the historical profile is retrieved from disk, the two are integrated, and the resulting profile is written back to disk.

Figure 13.19 shows `sig_main` for the Cell Tower application. This program takes four arguments: the directory storing the historical profiles, the new directory to contain the updated profiles, the location of the wireless call detail stream to be integrated, and

```
int getNext(pht_p p, char **s, int *lim, char delim, unsigned int *h){
  int i;     char t;     int result = HASH_OK;

  for(i=0; (i<*lim) && (delim != (*s)[i]); i++){};

  if (i) {
    t = (*s)[i];
    (*s)[i] = '\0';
    result = insert_pht(p,*s,h);
    (*s)[i] = t;
  } else *h = PHT_UNKNOWN;
  *s = *s + i + 1;
  *lim = *lim - (i+1);
  return result;
}

int getValidCall(char paramSet, pht_p p, Sfio_t *fp, wc_t* lcr){
  int l, result;     char *s;

  if (!paramSet) return HRS_STREAM_ERROR;    /* stream failure */

  result = sfscanf(fp, "%llu|%llu|", &(lcr->origin), &(lcr->dialed));
  if (2 != result) return HRS_STREAM_DROP_REC;

  if ((MINPN > lcr->origin) || (MAXPN < lcr->origin))
    return HRS_STREAM_DROP_REC;

  if ((MINPN > lcr->dialed) || (MAXPN < lcr->dialed))
    return HRS_STREAM_DROP_REC;

  s = sfgetr(fp, '\n', 1);     /* read rest of line: cell tower info */
  if (NULL == s) return HRS_STREAM_DROP_REC;
  l = sfvalue(fp);             /* number of bytes just read. */

  result = getNext(p, &s, &l, '|', &lcr->poct);
  if (HASH_OK != result) return HRS_STREAM_DROP_REC;

  result = getNext(p,&s, &l, '|', &lcr->soct);
  if (HASH_OK != result)  return HRS_STREAM_DROP_REC;

  result = getNext(p,&s, &l, '|', &lcr->pdct);
  if (HASH_OK != result)  return HRS_STREAM_DROP_REC;

  result = getNext(p,&s, &l, '|', &lcr->sdct);
  if (HASH_OK != result)  return HRS_STREAM_DROP_REC;

  return HRS_STREAM_KEEP_REC;                    /* valid record */
}
```

Figure 13.11: Translation function for wireless call detail stream.

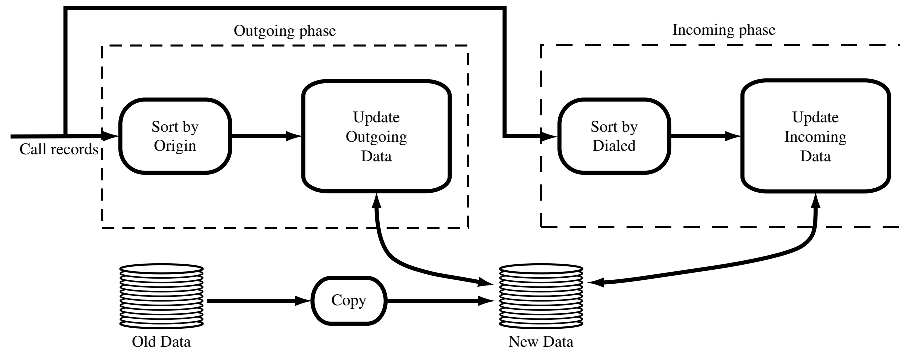


Figure 13.12: High-level architecture of signature computations. The processing typically consists of several phases, each sorting the data in a different order and updating a different part of the signatures.

```
void age(cellTower_m m){
  iterate( over m ){
       event phone(long long *k){
         int i;
         profile_t p = m<:*k:>;
         for (i=0;i<TOPN;i++){
           p.count[i] *= (1 - LAMBDA);    /* age existing data */
         }
         p.other *= (1 - LAMBDA);
         m<:*k:> = p;
     }
  }
}
```

Figure 13.13: Function that ages data in a Cell Tower map.

a string describing the date of the stream data. The function copies the historical data into the new data, ages the two profile collections, calls the functions `doOrigin` and `doDialed` to update the profile collections, and then sets the processing date.

```
void doOrigin(char *callsLoc, cellTower_m m, pht_p pht){
  wireless_s calls(:pht:) = callsLoc;
  profile_t p;
  iterate (
    over calls
    filteredby completeOriginCellCall
    sortedby origin
    withevents originDetect){

    event line_begin(long long origin){
      initProfile(&p);
    }

    event call(wc_t lcr){
      aggregate(&p, lcr, ORIGIN);
    }

    event line_end(long long origin){
      profile_t op2;
      profile_t op = m<:origin:>;
      integrate(&op,&p);
      m<:origin:> = op;
      op2 = m<:origin:>;
    }
  }
}
```

Figure 13.14: Function for updating outgoing Cell Tower map.

```
munion wline_e {: long long line_begin,
                  wc_t call,
                  long long line_end :}
```

Figure 13.15: Multi-union describing wireless call-detail events.

```
wline_e originDetect (wc_t *w[3:1]){
  wc_t *prev = w[0];
  wc_t *current = w[1];
  wc_t *next = w[2];
  wline_e b,e;

  if ((0 == prev) || (prev->origin != current->origin))
    b = {: line_begin = current->origin :};
  else
    b = (wline_e){: :};
  if ((0 == next) || (next->origin != current->origin))
    e = {: line_end = current->origin :};
  else
    e = (wline_e){: :};
  return b :+: {: call = *w[1] :} :+: e;
}
```

Figure 13.16: Event detection function for outgoing phone calls.

```
void doDialed(char *callsLoc, cellTower_m m, pht_p pht){
  wireless_s calls(:pht:) = callsLoc;
  profile_t p;
  iterate (
    over calls
    filteredby completeDialedCellCall
    sortedby dialed
    withevents dialedDetect){

    event line_begin(long long dialed){
      initProfile(&p);
    }

    event call(wc_t lcr){
      aggregate(&p, lcr, DIALED);
    }

    event line_end(long long dialed){
      profile_t op = m<:dialed:>;
      integrate(&op,&p);
      m<:dialed:> = op;
    }
  }
}
```

Figure 13.17: Function for updating incoming Cell Tower map.

```
wline_e dialedDetect (wc_t *w[3:1]){
  wc_t *prev = w[0];
  wc_t *current = w[1];
  wc_t *next = w[2];
  wline_e b,e;

  if ((0 == prev) || (prev->dialed != current->dialed))
    b = {: line_begin = current->dialed :};
  else
    b = (wline_e){: :};
  if ((0 == next) || (next->dialed != current->dialed))
    e = {: line_end = current->dialed :};
  else
    e = (wline_e){: :};
  return b :+: {: call = *w[1] :} :+: e;
}
```

Figure 13.18: Event detection function for incoming phone calls.

```
int sig_main(const exists cellTower_d oldCT <d:, "Old directory">,
             new cellTower_d newCT <D:, "New directory">,
             char *callsLoc <c:, "Wireless call detail">,
             char *date <t:, "Processing date">  default "unspecified" ) {

  newCT :=: oldCT;

  age(newCT->outMap);
  age(newCT->inMap);

  doOrigin(callsLoc,newCT->outMap, newCT->ctHashTable);
  doDialed(callsLoc,newCT->inMap, newCT->ctHashTable);

  newCT->lastUpdated = date;
}
```

Figure 13.19: Main program for the Cell Tower application.

Appendix A

Hancock 1.1 versus Hancock 1.0

The following sections describe how Hancock changed between versions 1.0 and 1.1.

Pickles

To integrate other data sources into Hancock and to provide support for variable-width data, we designed a new basic Hancock type: pickles. Chapter 8 describes this new type in detail.

Maps

To improve the efficiency of various patterns of access to data in maps, we changed the on-disk representation of maps. Consequently, maps built using previous versions of Hancock must be converted to the new representation to enable access from programs compiled in Hancock 1.1. Appendix A describes how to do the conversion.

Default compression functions for maps now produce platform-independent representations provided that the value type of the map uses the fixed-sized types described in Chapter 12.

Directories

To make the semantics of directories consistent with that of maps and pickles, we changed the in-memory representation of directories from structs to pointers to structs. Hence the following program in version 1.0:

```
directory dir_d {
  int myInt;
};
void sig_main(){
  dir_d d = "myDirectory";
  d.myInt = 10;
}
```

must be changed to the following in version 1.1:

```
directory dir_d {
    int myInt;
};

void sig_main(){
    dir_d d = "myDirectory";
    d->myInt = 10;
}
```

Only line 6 differs between these two code fragments; `d.myInt` became `d->myInt` in the second fragment.

A.1 Converting Hancock 1.0 Maps to Hancock 1.1 Maps

The representation used for maps in Hancock 1.0 is not compatible with the representation used in Hancock 1.1. The runtime system contains a function, `HRScvtOldMaptoNew`, that converts Hancock 1.0 maps into Hancock 1.1 maps. The following sample Hancock program uses this function to convert a usage map into the new format.

```
#include "map.hh"

int sig_main(new usage_m newUsage <U:>,
             exists const usage_m oldUsage <u:>)
{
   HRScvtOldMaptoNew(oldUsage, newUsage);
}
```

Note that both the old map and the new map must have the same Hancock type declaration.

The runtime system will generate an error if a program tries to use any of the Hancock operators (other than the conversion function) on a Hancock 1.0 map.

Appendix B

Ranges

Older versions of Hancock supported a standard range type that allows programmers to specify that a value will be an integer from a particular interval. For example,

```
range npanxx_r = [0..(MAXNPANXX-1)];
```

defines a type, `npanxx_r`, with values that lie between 0 and `(MAXNPANXX-1)` inclusive.[1] The end points of a range definition must be constant C expressions.

It is legal to use a range type, such as `npanxx_r`, anywhere it is legal to use the C type `int`. Programmers can declare variables, parameters, and structure fields to have a range type, but they are used mainly in older versions of Hancock to define key types for maps. The compiler does not insert automatic range checking code at present. 1

[1] We use the convention that type names ending in an `_r` denote `ranges`.

Appendix C

ScampRec.hh

```
/* Header file for describing binary stream of call detail records. */
#ifndef __SCAMPREC_H
#define __SCAMPREC_H

#define MAXNPANXX 1000000
#define MAXNPA 1000
#define MAXNXX 1000
#define MAXLINE 10000
#define MAXBASE (MAXNXX*MAXLINE)
#define TELNUMSIZE 10

#define MINVALIDPN 2000000000LL
#define MAXVALIDPN 10000000000LL

#define extractNPA(pn) ((pn)/MAXBASE)
#define extractNXX(pn) (((pn)%MAXBASE)/MAXLINE)
#define extractNPANXX(pn) ((pn)/MAXLINE)
#define extractLINE(pn) ((pn)%MAXLINE)

/* Physical representation */
typedef struct {
  int onpa, obase;
  int dnpa, dbase;
  int con;
  int dur;
} pScampRec_t;

typedef long long pn_t;
typedef int areacode_t;
typedef int exchange_t;
typedef int Hdate_t;
typedef int Htime_t;
```

```
/* Logical representation */
typedef struct {
   pn_t origin;
   pn_t dialed;
   Hdate_t connecttime;
   Htime_t duration;
   char isTollFree;
   char isIncomplete;
   char isIntl;
} scampRec_t;

/* Translation function */
int getValidCall(pScampRec_t *p, scampRec_t *h)
{
  if (p->onpa < 200 || p->onpa > 999 )  return HRS_STREAM_DROP_REC;
  if ((p->obase < 0) || p->obase > 9999999) return HRS_STREAM_DROP_REC;

  h->origin = ((long long) p->onpa)*MAXBASE + p->obase;

  if (p->dnpa == -15) {          /* international */
    h->isIntl = 1;
    h->isTollFree = 0;
    h->dialed = 0;
  }
  else {                         /* not international */
    h->isIntl = 0;

    if ((p->dnpa  < 200) || (p->dnpa > 999))      return HRS_STREAM_DROP_REC;
    if ((p->dbase < 0)   || (p->dbase > 9999999)) return HRS_STREAM_DROP_REC;

    if ((p->dnpa == 800) || (p->dnpa == 888) ||
        (p->dnpa == 877) || (p->dnpa == 866) || (p->dnpa == 855))
      h->isTollFree = 1;
    else
      h->isTollFree = 0;

    h->dialed = ((long long) p->dnpa)*MAXBASE + p->dbase;
  }
  h->isIncomplete = (0 == p->dur);
  h->connecttime     = p->con;
  h->duration        = p->dur;
  return HRS_STREAM_KEEP_REC;
}

/* Stream declaration */
stream callDetail_s {
  getValidCall : pScampRec_t => scampRec_t;
};
```

```
munion line_e {: areacode_t npa_begin,
                 exchange_t nxx_begin,
                 pn_t line_begin,
                 scampRec_t call,
                 pn_t line_end,
                 exchange_t nxx_end,
                 areacode_t npa_end :};

/* Event detection code. */
line_e beginLineDetect(pn_t *prev, pn_t *current) {
  areacode_t a = extractNPA(*current);
  exchange_t n = extractNPANXX(*current);

  if ((prev==NULL) || (extractNPA(*prev) != a))
    return {: npa_begin = a, nxx_begin = n, line_begin = *current :};
  if (extractNPANXX(*prev) != n)
    return {: nxx_begin = n, line_begin = *current :};
  if (*prev != *current)
    return {: line_begin = *current :};

  return (line_e) {: :};
}

line_e endLineDetect(pn_t *current, pn_t *next){
 areacode_t a = extractNPA(*current);
 exchange_t n = extractNPANXX(*current);

  if ((next == NULL) || (a != extractNPA(*next)))
    return {: line_end = *current, nxx_end = n , npa_end = a :};
  if (n != extractNPANXX(*next))
    return {: line_end = *current, nxx_end = n :};
  if (*current != *next)
    return {: line_end = *current :};

  return (line_e) {: :};
}

line_e originDetect(scampRec_t *w[3:1])
{ line_e b,e;
  pn_t *prev, *curr, *next;

  prev = ((w[0] == NULL) ? NULL : &(w[0]->origin));
  curr = &(w[1]->origin);
  next = ((w[2] == NULL) ? NULL : &(w[2]->origin));
  b = beginLineDetect(prev,curr);
  e = endLineDetect(curr,next);

  return b :+: {: call = *w[1] :} :+: e;
}

#endif
```

Appendix D

WorldNet.hh

```
/* Header file describing general stream of WorldNet session records. */

#ifndef __SESSION_H
#define __SESSION_H

#define SECSINDAY   60 * 60 * 24
#define extractDATE(utime) (utime / SECSINDAY)
#define extractTIME(utime) (utime % SECSINDAY)

#define MIN_ID 100000000
#define MAX_ID 1000000000

typedef int userID_t;
typedef int Hdate_t;
typedef int Htime_t;
#define NAME_LENGTH 30

/* Logical representation of WorldNet session.*/
/* pop names have short bounded length,
 * so use fixed-width representation. */
typedef struct {
  userID_t id;
  char pop[NAME_LENGTH + 1];
  Hdate_t connect_date;
  Htime_t connect_time;
  int duration;
} session_t;
```

```
/* Translation function.  Each line represents one WorldNet session. */
int getValidSession(Sfio_t *input, session_t *s)
{
  userID_t id;
  char tpop[1000];
  int utime;
  int duration;
  int found;

  if ((found = sfscanf(input, "%d %s %d %d\n",
                  &id, &tpop, &utime, &duration)) == EOF)
    return HRS_STREAM_DROP_REC;

  if (found != 4)
    return HRS_STREAM_DROP_REC;

  if (strlen(tpop) > NAME_LENGTH)
    return HRS_STREAM_DROP_REC;
  else
    strcpy(s->pop, tpop);

  s->connect_date = extractDATE(utime);
  s->connect_time = extractTIME(utime);

  s->duration = duration;

  if ((id < MIN_ID) || (id > MAX_ID))
    return HRS_STREAM_DROP_REC;
  s->id = id;

  return HRS_STREAM_KEEP_REC;
}

/* General stream declaration. */
stream worldNet_s {
  getValidSession: Sfio_t => session_t;
};
```

```
/* Multi-union for describing events related to WorldNet sessions.   */
munion id_e {: userID_t id_begin,
               session_t rec,
               userID_t id_end :};

/* Auxiliary functions.   */
id_e beginIdDetect(userID_t *prev, userID_t *current)
{
  if ((prev==NULL) || (*prev != *current))
    return {: id_begin = *current :};

  return (id_e) {: :};
}

id_e endIdDetect(userID_t *current, userID_t *next)
{
  if ((next == NULL) || (*current != *next))
    return {: id_end = *current :};

  return (id_e) {: :};
}

/* Event detection function.   */
id_e idDetect(session_t *w[3:1])
{ id_e b, e;
  userID_t *prev, *curr, *next;

  prev = ((w[0] == NULL) ? NULL : &(w[0]->id));
  curr = &(w[1]->id);
  next = ((w[2] == NULL) ? NULL : &(w[2]->id));
  b = beginIdDetect(prev, curr);
  e = endIdDetect(curr, next);

  return b :+: {: rec = *w[1] :} :+: e;
}

#endif
```

Appendix E

ipRec.hh

```
/* Header file for describing general stream of tcpdump data,
 * includes stream for describing stream of ip addresses.      */

#define MAXOCT 256
#define MAXPRIMARY (MAXOCT*(MAXOCT-1)+(MAXOCT-1))

void longToDot(unsigned long l,
               unsigned char *v0, unsigned char *v1,
               unsigned char *v2, unsigned char *v3){
  *v0 = l % MAXOCT;    l = l / MAXOCT;
  *v1 = l % MAXOCT;    l = l / MAXOCT;
  *v2 = l % MAXOCT;    l = l / MAXOCT;
  *v3 = l % MAXOCT;
}

unsigned long dotToLong(unsigned char v0, unsigned char v1,
                        unsigned char v2, unsigned char v3){
  return v3 * MAXOCT * MAXOCT * MAXOCT +
         v2 * MAXOCT * MAXOCT +
         v1 * MAXOCT + v0;
}

/* Logical representation of ip address. */
typedef  struct {
  unsigned char v0;
  unsigned char v1;
  unsigned char v2;
  unsigned char v3;
  unsigned long hash_value;
} ipAddr_t;
```

```
/* Logical representation of tcmdump record. */
typedef struct{
  unsigned int ts1, ts2;
  unsigned char version;
  unsigned char tos;
  unsigned short length;
  unsigned short identification;
  unsigned short fragment;
  unsigned char ttl;
  unsigned char protocol;
  ipAddr_t source;
  ipAddr_t dest;
} ipPacket_t;

void eatToEOL(Sfio_t *input)
{ int c;
   while (((c = sfgetc(input)) != EOF) && (c != '\n'))
     ;
}

/* Translation function for general stream of tcmdump packets. */
int getValidIPPacket(Sfio_t *input, ipPacket_t *ipP)
{ int v0, v1, v2, v3;
  unsigned int c0, c1;
  unsigned int s0;
  int found;

  found = sfscanf(input, "%d.%d|IP|", &ipP->ts1, &ipP->ts2);
  if ((found == EOF) || (found != 2)) return HRS_STREAM_DROP_REC;

  found = sfscanf(input, "%d.%d.%d.%d|", &v0, &v1, &v2, &v3);
  if ((found == EOF) || (found != 4)) return HRS_STREAM_DROP_REC;
  if ((v0 < 0) || (v0 > MAXOCT) ||
      (v1 < 0) || (v1 > MAXOCT) ||
      (v2 < 0) || (v2 > MAXOCT) ||
      (v3 < 0) || (v3 > MAXOCT)) {
    sfprintf(sfstdout, "%d.%d\n", ipP->ts1, ipP->ts2);
    eatToEOL(input);
    return HRS_STREAM_DROP_REC;
  }
  ipP->source.v0 = v0; ipP->source.v1 = v1;
  ipP->source.v2 = v2; ipP->source.v3 = v3;
  ipP->source.hash_value = dotToLong(v0,v1,v2,v3);

     ... continued ...
```

```
      ... continued ...

  found = sfscanf(input, "%d.%d.%d.%d|", &v0, &v1, &v2, &v3);
  if ((found == EOF) || (found != 4)) return HRS_STREAM_DROP_REC;

  if ((v0 < 0) || (v0 > MAXOCT) ||
      (v1 < 0) || (v1 > MAXOCT) ||
      (v2 < 0) || (v2 > MAXOCT) ||
      (v3 < 0) || (v3 > MAXOCT)) {
    sfprintf(sfstdout, "%d.%d\n", ipP->ts1, ipP->ts2);
    eatToEOL(input);
    return HRS_STREAM_DROP_REC;
  }
  ipP->dest.v0 = v0; ipP->dest.v1 = v1;
  ipP->dest.v2 = v2; ipP->dest.v3 = v3;
  ipP->dest.hash_value = dotToLong(v0,v1,v2,v3);

  found = sfscanf(input, "%d|%d|", &c0, &c1);
  if ((found == EOF) || (found != 2)) return HRS_STREAM_DROP_REC;
  ipP->version = (char) c0;
  ipP->protocol = (char) c1;

  if (ipP->version != 4) {
    sfprintf(sfstdout, "%d.%d", ipP->ts1, ipP->ts2);
    eatToEOL(input);
    return HRS_STREAM_DROP_REC;
  }

  found = sfscanf(input, "%d|", &s0);
  if ((found == EOF) || (found != 1)) return HRS_STREAM_DROP_REC;
  ipP->length = (short) s0;

  found = sfscanf(input, "%d|", &s0);
  if ((found == EOF) || (found != 1)) return HRS_STREAM_DROP_REC;
  ipP->identification = (short) s0;

  found = sfscanf(input, "%d|", &c0);
  if ((found == EOF) || (found != 1)) return HRS_STREAM_DROP_REC;
  ipP->ttl = (char) c0;

  found = sfscanf(input, "%d|", &c0);
  if ((found == EOF) || (found != 1)) return HRS_STREAM_DROP_REC;
  ipP->tos = (char) c0;

  found = sfscanf(input, "%d|", &s0);
  if ((found == EOF) || (found != 1)) return HRS_STREAM_DROP_REC;
  ipP->fragment = s0;

  eatToEOL(input);
  return HRS_STREAM_KEEP_REC;
}
```

```
/* General stream declaration for tcpdump stream. */
stream ipPacket_s { getValidIPPacket : Sfio_t => ipPacket_t;};

/* Translation function for general stream of ip addresses. */
int getvalidIPaddr(Sfio_t *input, ipAddr_t *addr)
{
  int v0, v1, v2, v3, found;

  found = sfscanf(input, "%d.%d.%d.%d\n", &v0, &v1, &v2, &v3);
  if ((found == EOF) || (found != 4))
    return HRS_STREAM_DROP_REC;
  if ((v0 < 0) || (v0 > MAXOCT) ||
      (v1 < 0) || (v1 > MAXOCT) ||
      (v2 < 0) || (v2 > MAXOCT) ||
      (v3 < 0) || (v3 > MAXOCT)) {
    return HRS_STREAM_DROP_REC;
 }
  addr->v0 = v0;   addr->v1 = v1;
  addr->v2 = v2;   addr->v3 = v3;
  addr->hash_value = dotToLong(v0,v1,v2,v3);
  return HRS_STREAM_KEEP_REC;
}

/* General stream declaration for stream of ip addresses. */
stream ipAddr_s { getvalidIPaddr : Sfio_t => ipAddr_t;};
```