Testing Formal Semantics: Handel-C

by

Brian J. Corcoran, B.Sc.

Thesis

Presented to the

University of Dublin, Trinity College

in partial fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Brian J. Corcoran

12 September 2005

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Brian J. Corcoran

12 September 2005

Testing Formal Semantics: Handel-C

Brian J. Corcoran
University of Dublin, Trinity College, 2005

Supervisor: Andrew Butterfield

This dissertation addresses the formal semantics of Handel-C: a C-based language with true parallelism and priority-based channel communication, which can be compiled to hardware. It describes an implementation in the Haskell functional programming language of a denotational semantics for Handel-C, as described in (Butterfield & Woodcock, 2005a). In particular, the *Typed Assertion Trace* trace model is used, and difficulties in creating a concrete implementation of the abstract model are discussed. An existing toolset supporting a operational semantics for the language is renovated, in part to support arbitrary semantic "modes," and to add support for the denotational semantics using this feature. A comparison module is written to compare the traces of two programs in any semantic mode supported by the simulator.

Random testing support is implemented via the QuickCheck testing tool for Haskell. This tool is incorporated into the comparison module, allowing testing of various properties of Handel-C, as well as its traditional use of testing the Haskell implementation for errors. This support is used to search for discrepancies between the operational and denotational semantic models.

Finally, several proposed "Laws of Handel-C" are implemented and tested using the QuickCheck module. Some errors in the specification of the laws are discovered and corrected. Once the specifications are corrected, all of the proposed laws pass, paving the way for future formal verification.

Acknowledgments

I would like to thank my supervisor, Andrew Butterfield, for his help and support, and for introducing me to the wonderful worlds of CSP, formal methods, and Haskell.

Many thanks to the entire NDS class for a fantastic and unforgettable year in Ireland.

BRIAN J. CORCORAN

University of Dublin, Trinity College
September 2005

Contents

Abstract **iv**

Acknowledgments **v**

List of Tables **x**

List of Figures **xi**

Chapter 1 Introduction **1**

- 1.1 Motivation 1
- 1.2 Objectives 1
- 1.3 Document Structure 2

Chapter 2 Background **3**

- 2.1 Handel-C 3
  - 2.1.1 CSP-like features 3
  - 2.1.2 Timing 7
  - 2.1.3 `prialt` 7
  - 2.1.4 Other Differences 8
  - 2.1.5 Mini Handel-C Language 8
- 2.2 Existing work 10
  - 2.2.1 Typed Assertion Traces 10
  - 2.2.2 Existing Tool Support 11
- 2.3 Haskell 12
- 2.4 QuickCheck 13
  - 2.4.1 Defining Properties 13
  - 2.4.2 Generators 13

Chapter 3 Denotational Semantics Implementation **16**

- 3.1 Worlds, Changes, and Choices 17
  - 3.1.1 Handel-C Domains as Events 19

3.1.2 Functions on Worlds 19
3.1.3 `emerge` 21
3.2 Trace Implementation 23
3.2.1 TrcSet Data Structure 23
3.2.2 Guarded Events 23
3.2.3 Propositions 23
3.2.4 Predicates 25
3.3 Program Semantics 26
3.3.1 Null Process 26
3.3.2 Clock Tick 26
3.3.3 Assignment 26
3.3.4 Sequential Composition 27
3.3.5 Parallel Composition 27
3.3.6 Conditional 27
3.3.7 Iteration (While) 28
3.3.8 PriAlt 28
3.3.9 PriAlt-Request 29
3.3.10 PriAlt-Wait 30
3.3.11 PriAlt-Case 30
3.3.12 Useful Functions 31
3.4 Running and Stepping 33
3.4.1 Stepping 33
3.4.2 Running 35
3.4.3 Denotational Semantics State 36

Chapter 4 Handel-C Simulator **37**
4.1 Comparison Module 37
4.1.1 Compare Current Environment 37
4.1.2 Compare Traces 37
4.2 Simulator Changes 38
4.3 Problems with Original Tool 39
4.4 Semantic State 39
4.4.1 Type of semantics 39

4.4.2 Semantic state class 39
4.4.3 Semantic state abstract implementation 41
4.5 Multiple Files 42

Chapter 5 Handel-C QuickCheck Support **44**
5.1 QuickCheck Generators 44
5.1.1 Random Environments 44
5.1.2 Random Datum 45
5.1.3 Random Programs 45
5.1.4 Random Expressions 46
5.1.5 Random Guarded Events 47
5.2 Testing Trace Equivalence 50
5.2.1 Trace Equivalence (Op. Sem.) 50
5.2.2 Trace Equivalence (Den. Sem.) 51
5.2.3 Semantic Equivalence 51

Chapter 6 Results **53**
6.1 Typed Assertion Traces 53
6.1.1 Concatenating for Microslots 53
6.1.2 Merging two microslot-sequences in parallel 54
6.2 Traces 56
6.2.1 While 56
6.2.2 If 57
6.2.3 Sequential/Parallel Composition 58
6.2.4 Improving efficiency 59
6.3 Testing Semantic Equivalence 61
6.3.1 Opfail Example 61
6.4 Laws of Handel-C 63
6.4.1 Simple Example 64
6.4.2 Complex Example 64
6.4.3 Failing Example 65

Chapter 7 Conclusion **67**
7.1 Objectives fulfilled 67

7.2 Future work 68
    7.2.1 Simulator 68
    7.2.2 Overall project 68

Appendices **69**

Appendix A Laws of Handel-C **69**
A.1 Law Categories 69
    A.1.1 Testing Procedure 69
A.2 Structural Laws 69
A.3 Conditional Laws 73
A.4 Event Laws 74
A.5 `prialt` Laws 77

Appendix B HCSemTool Change Log **81**
B.1 HCAbsSyn 81
B.2 HCOpSem 81
B.3 HCState 82
B.4 TypedAssertionTraces 83

References **84**

List of Tables

2.1 Factorial: Sequential vs. Parallel 6
2.2 mini-Handel-C Statement Syntax 9

6.1 Interleave example 58
6.2 Profiling Interleave Modification 60
6.3 Timing operational semantics vs. denotational semantics 61

List of Figures

1.1 Project Overview 2

2.1 Handel-C Parallel Flow 4
2.2 Handel-C Channel Communication 6
2.3 QuickCheck Property Example 13
2.4 QuickCheck Test Example 14
2.5 QuickCheck Arbitrary Example 14

6.1 `opfail` Handel-C compiler error 63

Listings

2.1 `prialt` example 8
4.1 Revised PST 43
6.1 msglue 54
6.2 msspar 56
6.3 opfail.hcc 62

Chapter 1

Introduction

1.1 Motivation

The motivation for this project is to aid in the creation of a formal semantics for the Handel-C programming language. In particular, this work attempts to provide appropriate methodology and tool support toward providing a high level of assurance for Handel-C programs. The recent work done in this area is described in (Butterfield & Woodcock, 2005a).

1.2 Objectives

The objectives of this project are firstly to implement the denotational semantics of Handel-C, as specified in (Butterfield & Woodcock, 2005a). An existing toolset, currently supporting the operational semantics of Handel-C, will be extended to also support the simulation of the denotational semantics.

Next, a comparison module will be written to compare the results of the two semantic models. This module will be used to search for any discrepancies between the operational and denotational semantic models. To reach this end, experiments to test the two models will be devised and run through the toolset. The results of these experiments will be used to revise the semantic models, which in turn will be used to update the toolset. The updated toolset will used to re-run the experiments.

Finally, a tool will be written using the QuickCheck module to automatically test several proposed "Laws of Handel-C". For an overview of the project, see Fig. 1.1.

The end research aims are the verification or refutation of the hypothesis that various proposed formal semantics are correct and say the same thing. This work should also provide a prototype tool for formal reasoning about the Handel-C language, and serve as a baseline for future tool support.

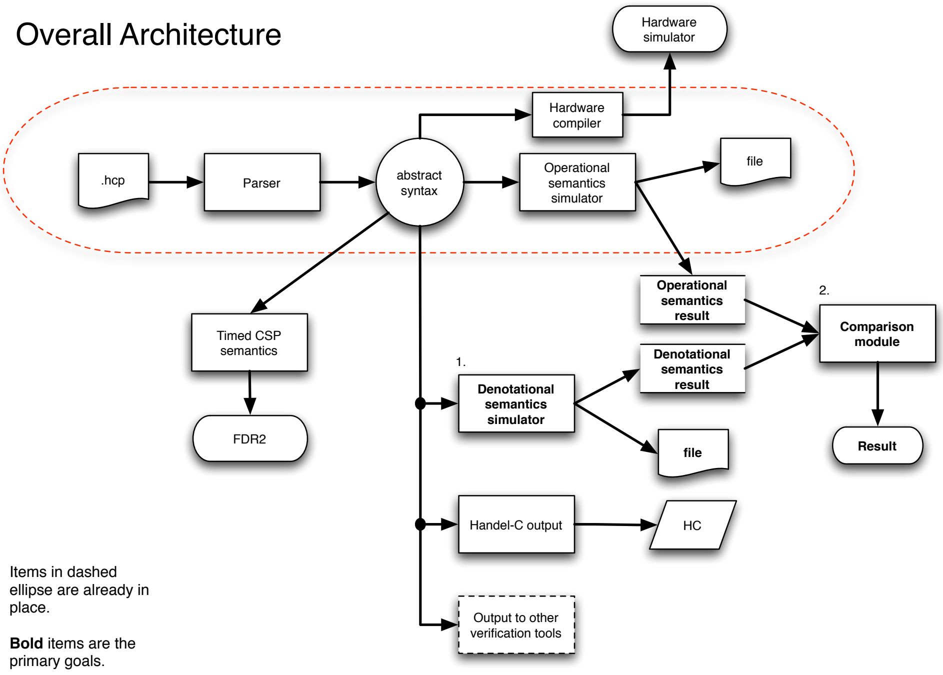

Figure 1.1: Project Overview

1.3 Document Structure

Chapter 2 gives all the necessary background for this project, including descriptions of the Handel-C language, and of the pre-existing tool this project is based upon.

Chapter 3 details the implementation of the denotational semantics, Chapter 4 details the modifications to the simulator, including the comparison module, and Chapters 5 details the automatic testing functions, based upon QuickCheck.

Chapter 6 discusses various results of this thesis, including problems with implementing the trace model, and the results of testing semantic equivalence and various proposed "Laws of Handel-C." The conclusion is given in Chapter 7.

Finally, the full "Laws of Handel-C" and test results are given in Appendix A, and Appendix B contains a list of all changes to the pre-existing tool modules.

Chapter 2

Background

2.1 Handel-C

Handel-C (Celoxica Ltd, 2002), (Celoxica Ltd, 2004) is a programming language developed by the Hardware Compilation Group at Oxford University Computing Laboratory, and now sold by Celoxica Ltd. It is ANSI-C based, with extensions based upon Concurrent Sequential Programs (CSP) (Hoare, 1985), such as parallelism and channel-based communication. The channel communication features a priority-based conflict resolution construct called `prialt`. Handel-C compiles directly to low-level hardware such as field-programmable grid arrays (FPGAs). To support such hardware, Handel-C features several extensions for dealing with data types of arbitrary widths.

2.1.1 CSP-like features

True parallelism

Handel-C uses true parallelism, where each parallel thread executes at exactly the same time as the other threads. This is in contrast to normal personal computers, where parallelism is implemented by time-slicing: quickly switching between parallel threads fast enough to appear simultaneous to the user. Handel-C achieves true parallelism by generating separate pieces of hardware for each branch, and executing the branches at exactly the same time in each piece of hardware.

Parallelism is implemented via the `par` construct. The `par` block signifies that execution should be split into a number of concurrent branches. Each branch is executed simultaneously, and the executions combine at the end of the `par` block. The execution can only continue once all the branches have finished; any branches that finish early must continually delay. This is shown in Fig. 2.1.

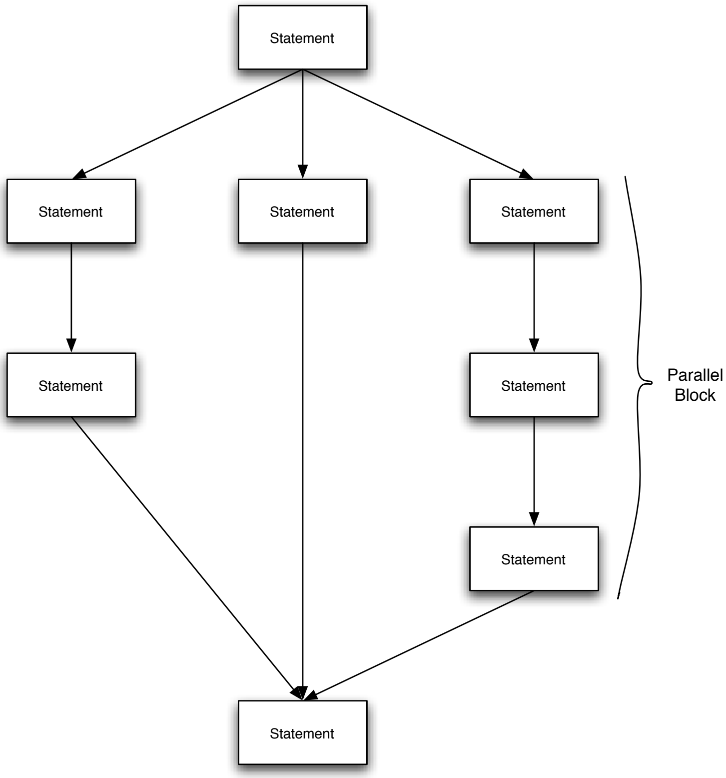

Figure 2.1: Handel-C Parallel Flow

Parallel access to variables

Handel-C allows variables to be shared between parallel branches; this leads to concurrency issues when accessing the variables. (Celoxica Ltd, 2004) offers two solutions to this issue, one stringent, and one more relaxed:

> The rules of parallelism state that the same variable must not be accessed from two separate parallel branches. This avoid resource conflicts on the variables.
>
> The rule may be relaxed to state that the same variable must not be assigned to more than once on the same clock cycle but may be read as many times as required.

An example of the power given by using the relaxed rules is the ability to swap two variables in a single clock step:

```
par
{
  a = b;
  b = a;
}
```

The Handel-C manual (Celoxica Ltd, 2004) also notes that since parallel assignment is run-time dependent, the Handel-C compiler is not able to check for all problems when the rules of parallelism are relaxed. This places the burden on the programmer, and represents an area where a formal semantics and tool support could be used to prove that such problems do not occur.

The use the of parallel construct is what gives Handel-C much of its power. The difficulty arises in finding places to use it. For example, examine the standard factorial function.

```
x = 5;
f = 1;
while (x > 1) {
    f = f * x;
    x = x - 1;
}
```

This function requires two clock steps for the initial assignments, plus two clock steps for each iteration, for a total of 10 steps, as seen in Table 2.1. However, if the programmer takes advantage of the parallel construct, this can be reduced to only 5 steps:

```
par {
    x = 5;
    f = 1;
}
while (x > 1) {
    par {
        f = f * x;
        x = x - 1;
    }
}
```

Factorial	**par-Factorial**
`x = 5;`	`x = 5 ‖ f = 1`
`f = 1;`	`f = 1 * 5 ‖ x = 5 - 1`
`f = 1 * 5;`	`f = 5 * 4 ‖ x = 4 - 1`
`x = 5 - 1;`	`f = 20 * 3 ‖ x = 3 - 1`
`f = 5 * 4;`	`f = 60 * 2 ‖ x = 2 - 1`
`x = 4 - 1;`	
`f = 20 * 3;`	
`x = 3 - 1;`	
`f = 60 * 2;`	
`x = 2 - 1;`	

Table 2.1: Factorial: Sequential vs. Parallel

Channel communication

Similarly to CSP, Handel-C uses *channels* to communicate between parallel processes. Channel communication is analogous to assignment between parallel branches, and likewise takes one clock step. In order to perform channel communication, one branch must be outputting data onto a channel at the same time another is reading from the channel. If one branch attempts to communicate before another, then it must wait until both become ready. In this way, channels also provide a method to synchronize parallel processes.

Fig. 2.2 shows two parallel branches communicating across a channel. Branch 2 is ready to output at point a, but branch 1 is not, so branch 2 must wait until the two are synchronized. When they are both ready (points b and c), data is sent from 2 to 1 via the channel.

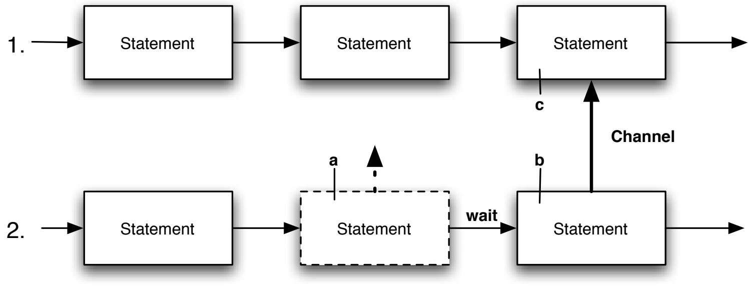

Figure 2.2: Handel-C Channel Communication

2.1.2 Timing

Since each `main` function in Handel-C has its own clock, timing is very precise. Statements in Handel-C are divided between those that take one clock cycle, and those that are “free”; that is, effectively take zero clock cycles. The statements that take one clock cycle are assignment, successful channel communication, and the `delay` command (which does nothing else). All conditionals, guards, and expression evaluation takes zero clock cycles.

In order to achieve this, however, the length of the clock cycle must be greater than the longest path through circuit logic; therefore it may be neccessary to limit how much evaluation is performed in a single step in order to run a particular clock speed. One result of having expressions take zero time is that Handel-C expressions cannot contain any side effects. However, the same effect can often be accomplish via use of parallel branches.

2.1.3 `prialt`

As mentioned previously, Handel-C allows for global variables, which can be accessed in any branch of execution. (Celoxica Ltd, 2004) states that “a single variable must not be written to by more than one parallel branch but my be read from by several parallel branches.[1]” The Handel-C solution is the `prialt` (priority alternatives) statement.

The `prialt` statement consists of a series of channels, in order of priority. In this paper, these offered channels are known as **guards**. They are tested sequentially until a ready channel is found, and that channel is given exclusive access. There is an optional “default guard,” which will chosen if no other channels are ready. If no default guard is present, and no other channels are active, the `prialt` will wait until the next clock step, and try again.

Each guard in a `prialt` (including the default guard) may have a statement associated with it. If channel is chosen, it performs its channel communication (one clock step) and then continues with is respective statement. If the default guard is chosen, it *immediately* executes its statement. This distinction proves to be very important in the implementation of the semantics.

An example `prialt` is given in Listing 2.1. This example first attempts to read from `chan1` into variable `y`. If `chan1` is not available, it attempts to read from `chan2` into `x`

---

[1] This is not strictly true, as a variables may be written from multiple parallel branches, as long as the writes occur at different clock steps.

Listing 2.1: `prialt` example

```
while(1)
    prialt
    {
        case chan1 ? y:
            break;
        case chan2 ? x:
            y = x*10;
            break;
        default:
            y = 0;
            break;
    }
```

and, if successful, assign `y` to `x`*10. Otherwise, it just sets `y` to zero.

2.1.4 Other Differences

Other major differences between Handel-C and ANSI-C are that Handel-C does not support standard floating point types such as float and double,[2] and integers can be any number of bits wide (including greater than 64 bits). Recursive functions are not allowed in Handel-C because the compiler must expand all logic before generating hardware. However, it is possible to use recursive macro expansions or recursive macro procedures to recurse a maximum number of times, or create multiple copies of a function. Finally, Handel-C cannot have an empty loop due to timing constraints. It is necessary to make sure that either the body of a loop will always execute at least once, or to provide an alternative via a conditional.

2.1.5 Mini Handel-C Language

Handel-C supports most of the ANSI-C standard, including arrays, multi-dimensional arrays, functions, pointers, structs, enums, and unions. In addition, it support extra syntax for dealing with hardware features such as clocks.

For the semantics of the language, a simplified, “mathematical” subset of the Handel-C language is being used. This subset represents all the major constructs of the language,

---

[2] Celoxica does provide a custom floating point library

and most constructs not present can be built by combining constructs in this subset. This "mini-Handel-C" is described in Table 2.2.

mini-Handel-C	**Handel-C**	**Description**
$p \in P \ ::= \ \mathbf{0}$	*none*	Null Statement
$\mid \ \mathbf{1}$	`delay`	Delay
$\mid \ x := e$	`x = e;`	Assignment
$\mid \ p_1;\ p_2$	`p1 ; p2`	Sequential Composition
$\mid \ p_1 \parallel p_2$	`par { p1; p2 }`	Parallel Composition
$\mid \ s \blacktriangleright [p_i]$	`select {...}`	Case statement
$\mid \ p_1 \lhd b \rhd p_2$	`if (b) {p1} else {p2}`	Conditional
$\mid \ b * p$	`while (b) {p}`	While
$\mid \ \langle g_i \rightarrow p_i \rangle$	`prialt {...}`	Prioritized Choice (`prialt`)

Table 2.2: mini-Handel-C Statement Syntax

As noted previously, assignment and delay take one clock cycle to execute, and all other statements (except `prialt`, detailed below) take "zero" time.

`prialt` notation

The `prialt` notation, $\langle g_i \rightarrow p_i \rangle$, is shorthand for $\langle g_1 \rightarrow p_1, \ldots g_n \rightarrow p_n \rangle$, where i represents the index from 1 to n. This syntax represents a prioritized sequence of guard-process pairs. Guards are members of the set:

$$g \in G ::= c?v \mid c!e \mid !?$$

where $c?v$ means an input guard on channel c to variable v, $c!e$ represents an output guard on channel c from expression e, and $!?$ represents the default guard. p_i represents any process. Note that a guard with no process is represented as $g \rightarrow \mathbf{0}$, and a non-`prialt` channel communication is represented as a `prialt` with only one element ($\langle g \rightarrow p \rangle$). As an example, Listing 2.1 on page 8 would be represented by:

$$\begin{array}{cllc} \langle & \mathit{chan}1?y & \rightarrow \mathbf{0}, & \\ & \mathit{chan}2?x & \rightarrow y := x * 10, & \\ & !? & \rightarrow y := 0 & \rangle \end{array}$$

2.2 Existing work

This work is part of a larger project being conducted by Dr. Butterfield in collaboration with Jim Woodcock (University of York, UK).

(Butterfield, 2001b) and (Butterfield, 2001a) represent an initial attempt at the operations and denotational semantics, respectively, of Handel-C, ignoring the `prialt` construct. The exclusion of `prialt` was due to a lack of documentation of how it actually worked. The semantics of the `prialt` statement were specifically examined in (Butterfield & Woodcock, 2002b). A complete operational semantics of Handel-C, including `prialt`, is given in (Butterfield & Woodcock, 2005b).

An initial denotational semantics, using traces based upon *Sequence-Trees*, is given in (Butterfield & Woodcock, 2002a). The semantics was later modified in (Butterfield & Woodcock, 2005a); in part to use *Typed Assertion Traces*.

2.2.1 Typed Assertion Traces

In the *Typed Assertion Trace* model of traces, each possible trace consists of a non-empty sequence of slots, where a slot represents a clock cycle. The denotational semantics maps a program to a set of such traces.

$$\begin{aligned} \tau \in \mathit{Trc} &= \mathit{Slot}^{+} \\ [\![\_]\!] &: P \rightarrow \{\, \mathit{Trc} \,\} \end{aligned}$$

A single trace consists of a sequence of guard/events pairs (described below). A set of traces contains all possible traces for a program. When a program is actually run, each trace in the trace set is compared to the current environment. Whenever the current guard is found to be false with respect to the current environment, the trace containing that guard is removed (pruned) from the trace set. At the end of running a (finite) program, all the traces in a trace set except one will have been pruned, with the remaining trace representing the actual trace of the program.

Each slot consists of two parts; the first is a series of guard/event pairs which determine if the a given trace is valid or not (called a microslot sequence, MS^{*}), and the second is a lone guard/event pair, where the event represents the actual clock step action (e.g., assignment), and is known and the $action$ event, and is given type act.

$$s, (\mu, a) \in \mathit{Slot} = \mathit{MS}^* \times \mathit{GE}$$

Each microslot consists of a triple of optional guarded events (s, q, r), which are of types $sel, req,$ and res, respectively.

$$m, (s, q, r) \in \mathit{MS} = \mathit{GE}^3$$

The sel type represents whether a given trace is valid because of a conditional or looping construct. Its guard represents one choice in a branch, and its event is always $null$.

The req and res types only deal with `prialts`. When a `prialt` is encountered, it registers all of its guards during the req phase. Between the req and res phases, the active guard (if any) is determined via the $Resltn$ function. During the res state, each trace checks to see if it has been activated, and the non-activated traces are pruned.

The reason that there is a sequence of microslots is the default guard. If no channel becomes active during the res phase, and a default guard exists, it is chosen. However, as mentioned previously, the default guard does not take a clock-cycle. Therefore the $sel - req - res$ loop is restarted. The loop only ends and progresses to the act event when a non-default guard is chosen, or when no default guards are present.

2.2.2 Existing Tool Support

This work builds off a pre-existing tool (Butterfield, 2005) written by Dr. Butterfield. This tool, written in the Haskell[3] programming language, supports the operational semantics and a "hardware compilation" semantics, including `prialt` support. The tool also includes a parser for an ASCII version of the mini-Handel-C syntax, and various function for pretty-printing output. semantics. The hardware semantics is not part of the scope of this project; however, it should be simple to add support for it; this is discussed in Section 4.4.

The operational semantics simulator allowed a file to be loaded or input by hand, initialized, then stepped through. This tool was used in the preparation of (Butterfield & Woodcock, 2005b), and proved to be helpful in determining/debugging the operational semantics described in that work:

---

[3] Haskell is described in Section 2.3

> Developing this [toolset] and experimenting with it helped correct minor problems with the operational semantics, as well as exploring various different types of rules, and variations in how transition types are calculated. In particular, the details of the transition condition table [. . .] were revised after such simulation experiments. All the examples in this paper have been simulated using the tool.

The primary goal of this project is extending this simulator with support for denotational semantics, comparing different files/semantics, adding automatic testing support, and general usability improvement. Ideally, this tool will prove to be as useful in writing future papers.

2.3 Haskell

All the code in this project is written in the *Haskell* programming language (*Haskell 98 language and libraries: The revised report*, 2002). The main features of Haskell are that it is:

- Strongly typed
- Lazy
- Purely functional
- Mathematics-based (syntax)

The primary reason for choosing this language is to benefit from the existing simulator code base, which was written in Haskell. However, there are many other reasons why Haskell is an appropriate language for this project. First, the denotational semantics deals with infinite data structures; in particular, traces are infinite whenever a loop is encountered. The laziness of Haskell allows these data structures to be evaluated only when needed, which greatly simplifies their design.

The mathematics-based syntax of Haskell allows a very close mapping between the proposed semantics and the actual code. This makes the code easier to implement as well as easier to understand.

Finally, Haskell has a very power testing tool, QuickCheck (discussed in Section 2.4), which is used for both testing the implementation for bugs, as well as attempting to prove various properties about the implementation of Handel-C. It could almost be said that the QuickCheck component of this project is an implementation of QuickCheck for Handel-C.

2.4 QuickCheck

QuickCheck (Claessen & Hughes, 2000) is a testing tool for Haskell programs. It works by formulating properties about a program, using formal specification to create a set of tests for functions. The specification language is implemented as Haskell functions, using the Haskell class system.

QuickCheck attempts to verify each property by testing it in "a large number[4]" of test cases. These test cases are generated automatically by random testing. In order to ensure that the random test distributions are meaningful, QuickCheck provides a "test data generation language," written in Haskell. It is possible to observe the distribution of test cases created, as well as restrict the cases created via invariants.

2.4.1 Defining Properties

Properties are defined using the QuickCheck *Property* type. For example, a property which states that the square of all integers is positive is show in Listing 2.4.1. It defines a property that, for any random Int n, $n^2 \geq 0$.

```
prop_PosSq :: Int → Property
prop_PosSq n = n*n ≥ 0
```

Figure 2.3: QuickCheck Property Example

This property can be tested in a Haskell interpreter by using the *test* command, as shown in Fig. 2.4.1. The test commands generates 100 random integers using the standard Int generator, and tests the property using each one.

2.4.2 Generators

Arbitrary

---

[4] By default, 100

```
QuickCheck> test prop_posSq
OK, passed 100 tests.
```

Figure 2.4: QuickCheck Test Example

QuickCheck provides standard generators for Ints, Lists, Strings, etc. However, any data type that implements *Arbitrary* monad can be used in properties to automatically generate data. An example of a Arbitrary for a 2-D coordinate is given in Fig. 2.4.2.

```
newtype Coord = Coord (Int,Int)

instance Arbitrary (Coord) where
newtype Coord = Coord (Int,Int)

instance Arbitrary (Coord) where
    arbitrary = do x ← arbitrary –– random x coord
                   y ← arbitrary –– random y coord
                   return (Coord (x,y))
```

Figure 2.5: QuickCheck Arbitrary Example

Custom Generators

In addition to generators for a specific datatype, custom generators can be defined. For example, the generator for Ints gives a wide range of values, but it may be necessary to generate Integers limited to certain restricts, in a certain distribution, or according to a specified parameter. This is done by creating a function which returns a `Gen` monad. QuickCheck supplies many functions to do this, such as choose, which picks a random value from a range. This is demonstrated in getDigit function, which generates a random base-10 digit.

```
genDigit :: Gen Int
genDigit = choose (0,9)
```

Generators are used in properties via the forAll command.

```
prop_Digit
 = forAll genDigit $ n → n < 10
```

Generators can be restricted by arbitrary invariants, as shown below:

```
prop_sumPos :: Int → Int → Bool
prop_sumPos m n
  = m > 0 ⟹
    n > 0 ⟹
    m+n > 0
```

When a property fails, it presents the failed counter example.

```
prop_PosSum :: Int → Bool
prop_PosSum n = n+n ≥ 0

QuickCheck> test prop_PosSum
Falsifiable, after 4 tests:
−1
```

Chapter 3

Denotational Semantics Implementation

The implementation of denotational semantics for Handel-C consists of four main sections:

Worlds, Changes, and Choices Section 3.1 is concerned with functions which handle the environment of the denotational semantics.

Traces An implementation of `Typed Assertion Traces`, described in Section 3.2.

Actual Semantics Section 3.3 describes a function to translate a Handel-C program into a set of Traces.

Running Functions to *prune* a set of Traces into an environment, detailed in Section 3.4.

The overall design plan was to write a simulator for the denotational semantics of a very simple imperative language first. This language would then be extended with additional functionality, until at some point it would be merged with the existing code base. The other possibility considered was to to start with the existing code base, and directly build upon it.

Overall, the decision to start from a simplified language was positive. Most importantly, it was a more accessible starting point; considering that it took a long time to "come up to speed" on the project, being able to start working with actual code was a great benefit. In particular, it wasn't necessary to have a detailed understanding of what all the existing code did, and why things were implemented how they were. It was still important, however, to browse the code and have a general idea of how it worked in order to be prepared for the eventual union of the two pieces.

Another benefit of starting with a simplified language is that it gives a much better understanding of the denotational semantics—there was already a proposed denotational semantics for Handel-C which was supposed to be implemented. By starting from scratch, there was an opportunity to make some of the same design decisions which had already been made. For example, traces were originally implemented as a function from one state to another;

however, once a parallel construct was added to the language this was no longer sufficient, and the Worlds/Changes/Choices model from the proposed semantics was adopted.

The biggest problem encountered was some amount of code with duplicate functionality. Much of this code made it into other parts of the project, however; some features of the simple language simulator were adopted into the existing simulator, and support for boolean types was moved over into the operational semantics. Other parts were simply discarded, such as an implementation of the Resolution function; however, having to implement this function was crucial to understanding how it works.

There were also some issues related to integrating the simple language code (which eventually grew to implement most of Handel-C's non-`prialt` functionality). These problems would have been encountered earlier if the alternative approach were taken; before there was a level of confidence with the language and methodology that greatly aided in resolving problems. Still, some functionality was difficult to integrate at a later stage; for example, the evalE function for evaluating expressions seemed to be fundamentally incompatible at first, but was eventually adopted (this was important so that the two semantics could share error messages).

Some functionality was not brought over from the simple language; in particular, the simple language implementation had the ability to choose from three trace models; a tree-based trace, a guarded trace model, and the full Typed Assertion Traces. In order to focus on getting one trace model working for the full semantics, support for multiple trace models were removed. However, the same technique used to implement multiple trace models was used to implement different *semantic states* for the simulator; these are detailed in Section 4.4.

3.1 Worlds, Changes, and Choices

In the denotational semantics, there are two types of environments; one which contains all the Ids of a program p (a *World*), and one which contains only those Ids which have changed in particular step (a *Change*). Both are implemented using the standard Environment.

$$\begin{aligned} \omega \in World_p &\;\widehat{=}\; \text{pIds}[\![p]\!] \to Datum \\ \delta \in Change_p &\;\widehat{=}\; \text{pIds}[\![p]\!] \xrightarrow{m} Datum \end{aligned}$$

```
type World  = Env NullAttr
type Change = Env NullAttr
```

Worlds and changes must be treated differently in order to handle parallel and a sequential composition with shared variables; functions cannot simply be composed (with the output environment of one function serving as the input environment for another. This is because the parallel functions must all have the same input at the beginning of a clock cycle, and only merge their output after they have all completed.

An example of this is the following:

$$P = x := -1 \triangleleft x > 0 \triangleright \mathbf{0}$$
$$Q = y := 1 \triangleleft x \leq 0 \triangleright y := 2$$
$$Q \mid\mid P$$

If these functions are composed as $Q \circ P$ with an initial environment of $x \mapsto 1$, the mappings will be:

$$\{\, x \mapsto 1, y \mapsto ? \,\} \rightarrow P \rightarrow \{\, x \mapsto -1, y \mapsto ? \,\} \rightarrow Q \rightarrow \{\, x \mapsto -1, y \mapsto 1 \,\}$$

However, the result should be:

$$\{\, x \mapsto 1, y \mapsto ? \,\} \rightarrow P \rightarrow \{\, x \mapsto -1 \,\}$$
$$\{\, x \mapsto 1, y \mapsto ? \,\} \rightarrow Q \rightarrow \{\, y \mapsto 2 \,\}$$

The various changes can then be merged together with the original environment to form the new environment. These mappings from worlds to changes are referred to as “choices”. It is also necessary to be able to identify the null choice.

$$\kappa \in Choice \quad \widehat{=} \quad World \rightarrow Change$$

```
data Choice = Choice (World→Change)
            | NullChoice
```

3.1.1 Handel-C Domains as Events

Choices play the role of "events" in the denotational semantics. The null event is the null choice, while event merging consists of merging each individual "change" from the choices.

```
instance Event Choice where
    (<>)  = emerge
    enull = NullChoice
```

Events are often compared to null in *Typed Assertion Traces*; to facilitate this, Events must implement the `Eq` class. This test it only returns true if both Choices are null; all other Choices are never equal.

```
instance Eq Choice where
    NullChoice == NullChoice = True
    _          == _          = False
```

Since Choices are implemented as datatypes, a function is needed to access the function the Choice contains. The apply function applies a choice to a world to generate a choice:

$$apply : Choice \to World \to Change$$

```
apply :: Choice → World → Change
apply NullChoice  w = nullChg
apply (Choice ch) w = ch w
```

apply is used to define a State:

```
instance State Choice World where
    stChg ch w = apply ch w
```

3.1.2 Functions on Worlds

Although Worlds and Changes currently have the same implementation, they are separated for consistency. Here are functions to create initial states; one for Worlds, and one for Changes.

```
nullWorld :: World
nullWorld = nullMap
```

```
nullChg :: Change
nullChg = nullMap
```

Update world

updateW applies a partial Change to an original World, to generate a new World.

$$updateW : Change \rightarrow World \rightarrow World$$

```
updateW :: Change → World → World
updateW ch w = override w ch
```

Get clock-step

Return the current clock-step from a World, as an integer.

$$getTau : World \rightarrow Int$$

```
getTau :: World → Int
getTau w = t
    where Dtime t = mApp w idTau
```

Tick

Perform a clock-tick on a World, to produce a new World with τ increased by one.

$$tick : World \rightarrow World_{\tau \mapsto \tau+1}$$

```
tick :: World → World
tick env = iMap idTau (Dtime (t + 1))
    where Dtime t = env!idTau
```

3.1.3 emerge

Event-merge (emerge) serves as the implementation of $\Diamond$ for Choices. It merges two Choices by removing all intersecting variables, taking the union of the independent variables, then attempting to resolve the conflicting variables.

This function slightly complicated by the fact that map conflicts in both communication events ($\Re$) and clock-tick events (τ) must be treated differently from standard assignment; these different behaviors are handled by the dmerge function. emerge first merges the events which have independent domains, then calls `dmerge` to resolve variable conflicts, which are then merged in.

$$
\begin{aligned}
\Diamond \quad &: \quad \mathit{Evt} \rightarrow \mathit{Evt} \rightarrow \mathit{Evt} \\
\oslash \Diamond e \quad &= \quad e \\
e \Diamond \oslash \quad &= \quad e \\
e_1 \Diamond e_2 \quad &= \quad \lambda\,\rho \rightarrow [e_1\rho \cup e_2\rho - (e_1\rho \cap e_2\rho)] \\
& \qquad \cup \mathit{mergeDup}(\mathrm{dom}\ e_1 \cap \mathrm{dom}\ e_2, e_1\rho, e_2\rho)
\end{aligned}
$$

```
emerge :: Choice → Choice → Choice
emerge p NullChoice = p
emerge NullChoice q = q
emerge (Choice p) (Choice q) = Choice ch
    where ch w = pch′ `mextend` qch′ `mextend` mrg
              where pch = p w
                    qch = q w
                    [pch′,qch′] = map (mremove dups) [pch,qch]
                    dups = dom (pch `mintersect` qch)
                    mrg = mergeDup dups pch qch
```

Merge duplicates

mergeDup merges a set of variables which have conflicts. The actual conflict is resolve via dmerge.

```
mergeDup :: Set Var → Change → Change → Change
mergeDup s p q
    | isNullSet s = nullChg
    | otherwise   = iMap v r `mextend` mergeDup (elems vs) p q
   where (v:vs) = ssort s
         r = dmerge (mApp p v) (mApp q v) v
```

Merge data values

dmerge merges two conflicting datum types. The `Id` parameter is used to give a more verbose error string. dmerge supports the following datum types:

Dint Updating an integer variable simultaneously returns an error

Dbool Same as Dint

Dundef Merging two undefined values is always the undefined value

Dtime Merging two time changes becomes a single clock-step.

Dpgr Merging two *PriGrp*s is simply their union

```
dmerge :: (Show a,Ord a) ⇒ Datum a → Datum a → Id → Datum a
dmerge (Dundef "") (Dundef "") _ = Dundef ""
dmerge (Dundef s1) (Dundef s2) _ = Dundef (s1++";"++s2)
dmerge (Dtime t1)  (Dtime t2)  _ = Dtime (max t1 t2)
dmerge (Dint n1)   (Dint n2)   v = Dundef (conflict v)
dmerge (Dbool _)   (Dbool _)   v = Dundef (conflict v)
dmerge (Dres (m1,s1)) (Dres (m2,s2)) _ = Dres (m1`mextend`m2,s1`union`s2)
dmerge d1 d2 v = Dundef ("Can′t␣merge␣types␣with␣values:␣′"
                      ++ show d1 ++ "′,␣′"
                      ++ show d2 ++ "′␣for␣variable␣′" ++ show v ++
                      "′")

conflict v = ("Cannot␣assign␣to␣same␣variable␣′"
             ++ v
             ++ "′␣in␣multiple␣processes.")
```

3.2 Trace Implementation

3.2.1 TrcSet Data Structure

First, the abstract Typed Assertion Trace is implemented using Expressions and Choices:

```
type TATi = TAT.Trc (SExpr NullAttr) Choice
```

The "TATi" type represents a single possible trace in all the outputs of the a program. All of these possible traces are combined to form the **TrcSet** data type.

The TrcSet data type is simply a list of instantiated Typed Assertion Traces:

$$TrcSet = TATi^{+}$$

```
data TrcSet = MkT [TATi]
```

Showing a TrcSet

Many programs (any containing a while loop or a `prialt` without a default guard) have infinite possible traces; therefore displaying a TrcSet must be restricted. By default, only the next ten traces are shown:

```
instance Show (TrcSet) where
    show tr = showN 10 tr
```

3.2.2 Guarded Events

Guarded events are instantiated as Expression, Choice tuples.

```
instance GrdEvt (SExpr NullAttr) Choice where
    genil  = (true,enull)
    gevoid = (false,enull)
```

3.2.3 Propositions

Propositions are implemented as boolean Expressions. However, propositions are compared on structural equality, not on logical equality. This means that, by default, $notp\ true \neq false$!

In particular, this caused a major problem in the ◇ (gemerge) operator from the *Typed Assertion Traces*:

```
gemerge :: (Prop p, GrdEvt p e) ⇒ GE p e → GE p e → GE p e
(p1,e1) `gemerge` (p2,e2)
    = (p1 &&& p2, e1 <> e2)
```

At a later part in the code, `p1 &&& p2` would be compared to `false`, which would always return $true$. For some cases, there is no easy solution to this problem, because propositions can include variables which can only be resolved at run-time. However, the common cases can be handled by reducing the combinatorial logic, which is what the function simplify does.

```
instance Prop (SExpr NullAttr) where
    false     = SBool False NA
    true      = SBool True  NA
    notp p    = simplify $ snot p
    p1&&&p2   = simplify $ p1 `sand` p2
    p1|||p2   = simplify $ p1 `sor` p2
```

Simplify

$$simplify : Expr \to Expr$$

Reduce the prepositional logic as far as possible, without knowing the current environment. For example, $(x > 0) \lor$ FALSE is simplified to FALSE (by the definition of $\lor$) but $(x > 0) \land$ FALSE is not reduced, because the state of x is unknown.

```
simplify :: SExpr NullAttr → SExpr NullAttr
simplify s@(SApp ((SVar op _):args) _)
    | op=="!"   = simp_not x
    | op=="&"   = simp_and x y
    | op=="|"   = simp_or  x y
    | otherwise = s
    where (x:_)    = args′
          (_:y:_)  = args′
          args′  = map simplify args
simplify s = s
```

```
simp_not p
    | p==strue              = sfalse
    | p==sfalse             = strue
    | otherwise             = snot p
p1 `simp_and` p2
    | p1==sfalse||p2==sfalse = sfalse
    | p1==strue&&p2==strue   = strue
    | otherwise              = p1 `sand` p2
p1 `simp_or` p2
    | p1==strue||p2==strue   = strue
    | p1==sfalse&&p2==sfalse = sfalse
    | otherwise              = p1 `sor` p2
```

3.2.4 Predicates

Predicates are implemented via Expressions and Worlds, and return a boolean value.

```
instance Predicate (SExpr NullAttr) World where
    assert p s = getBool ((evalE s) p)
```

3.3 Program Semantics

This section provides an implementation of the denotational semantic of Handel-C. The semantics are implemented in the function `sem` from Programs (SStmt) to a set of traces (TrcSet).

$$\llbracket \_ \rrbracket : P \rightarrow \mathsf{set}\ \mathit{Trc}$$

```
sem :: SStmt NullAttr → TrcSet
```

3.3.1 Null Process

The null process maps to a set containing the null trace.

$$\llbracket \mathbf{0} \rrbracket \ \widehat{=} \ \{\, \langle (\cdot) \rangle \,\}$$

```
sem (Sdelay 0 _) = MkT [tnull]
```

3.3.2 Clock Tick

A clock tick increments τ during the *act* phase.

$$\llbracket \mathbf{1} \rrbracket \ \widehat{=} \ \{\, \langle \llbracket \rangle \,\}$$

```
sem (Sdelay 1 _) = MkT [mkt Tact (true, tick′)]
    where tick′ = Choice (λw → tick w)

sem (Sdelay n a) = MkT [(mks Tact (true, tick′)) $: head rest]
    where (MkT rest) = sem (Sdelay (n−1) a)
          tick′ = Choice (λw → tick w)
```

3.3.3 Assignment

Assignment updates a variable, as well increments τ, during the *act* phase.

$$\llbracket x := e \rrbracket \ \widehat{=}\ \{\, \langle (\langle\rangle, \llcorner x \mapsto (\!| \, e \, |\!) \lrcorner) \rangle \,\}$$

```
sem (Sassign v e _) = MkT [mkt Tact (true,ch)]
    where
    ch = Choice (λw → override (tick w) (iMap v (evalE w e)))
```

3.3.4 Sequential Composition

Sequential composition must concatenate every trace from p with every trace from q. Since they are both possibly infinite, care must be taken to ensure that the resulting trace set can be searched effectively. This is accomplished via the mergeTS function, which is detailed in Section 3.3.12; this function takes an argument that links two traces, which in this case is the $\fatsemi$ (trace concatenation) function.

$$\llbracket p;\ q \rrbracket \ \widehat{=}\ \llbracket p \rrbracket \,\{\, \fatsemi \,\}\, \llbracket q \rrbracket$$

```
sem (Sseq ps _) = mergeTS (+++)  (map sem ps)
```

3.3.5 Parallel Composition

The implementation of parallel composition is almost identical to sequential composition, but individual traces are combined using the $\square\square$ operator, which merges two traces in parallel. It also uses mergeTS to deal with merging the two infinite sets of traces.

$$\llbracket p \parallel q \rrbracket \ \widehat{=}\ \llbracket p \rrbracket \,\{\, \square\square \,\}\, \llbracket q \rrbracket$$

```
sem (Spar ps _) = mergeTS (|!|)  (map sem ps)
```

3.3.6 Conditional

The conditional statement generates all the possible traces for when b is true, (`aff`) and for when it is false (`neg`). Since these two sets of traces may be infinite, they must be shuffled so that the traces are tried alternatively. This is accomplished via the shuffle function, described in Section 6.2.2.

$$[\![p \lhd b \rhd q]\!] \;\widehat{=}\; ((\!|\, b \,|\!), \varnothing)\,;^{\{\,\}}_{sel}\, [\![p]\!] \;\cup\; ((\!|\, \neg\, b \,|\!), \varnothing)\,;^{\{\,\}}_{sel}\, [\![q]\!]$$

```
sem (Sif b p q _) = MkT (shuffle aff neg)
    where MkT aff = tmap (($:) aslot) (sem p)
          MkT neg = tmap (($:) nslot) (sem q)
          aslot = mks Tsel (b,enull)
          nslot = mks Tsel (notp b,enull)
```

3.3.7 Iteration (While)

The implementation of while also must be aware of infinite traces. The key idea is that the traces must be tried in order of increasing size, so that the smallest trace is tried first. This is accomplished by always attempting the negative (`fin`) case before the looping case. The while loop is implemented in the *sel* stage of the traces.

$$\begin{array}{lcl} [\![b * p]\!] & \widehat{=} & \mathsf{fix}\,\mathcal{W}, \qquad b \not\equiv w\langle g_i \rangle \\ \textbf{where} & & \\ \mathcal{W}(T) & = & \{\, \langle mk_{sel}((\!|\, \neg\, b \,|\!), \varnothing) \rangle \,\} \cup ((\!|\, b \,|\!), \varnothing)\,;^{\{\,\}}_{sel}\, ([\![p]\!] \,\{\fatsemi\}\, T) \end{array}$$

```
sem (Swhile b p _) = tcons fin (MkT step)
    where fin = mkt Tsel ((notp b),enull)
          MkT step = tmap (($:) sslot) (sem cont)
          sslot = mks Tsel (b,enull)
          cont = sseq [p, (swhile b p)]
```

3.3.8 PriAlt

`prialt` is not given a semantics directly; instead it is translated into “pseudo-statements” (statements that do not actually exist in the Handel-C language). These statements are:

1. Submitting a request ($^{+}\langle g_i \rangle$);
2. Waiting until the request becomes active, and re-submitting the request on every clock cycle until it does ($wait\langle g_i \rangle$);
3. Selecting and executing the active guard and corresponding process ($a\langle g_i \rangle \blacktriangleright [\mathsf{act}(g_i);\ p_i]$)

$$\langle g_i \rightarrow p_i \rangle \quad = \quad {}^{+}\langle g_i \rangle;\ wait\langle g_i \rangle;\ a\langle g_i \rangle \blacktriangleright [\mathsf{act}(g_i);\ p_i]$$

($\langle g_i \rightarrow p_i \rangle$ is shorthand for $\langle g_1 \rightarrow g_1, \ldots, g_n \rightarrow p_n \rangle$ where i is assumed to index over $1 \ldots n$ for appropriate n.)

```
sem (Sprialt gps _)
 = sem (sseq [req,wait,select])
    where req     = Sreq gs NA
          wait    = Swait gs NA
          select  = Scond (Ssel gs NA) ps NA
          (gs,ps) = unzip gps
```

We can now give the semantics of the additional `prialt` constructs:

3.3.9 PriAlt-Request

The `prialt` request statement occurs during the req phase, and sets the b component in the environment.

$$[\![{}^{+}\langle g_i \rangle]\!] \quad \widehat{=} \quad \{\, \langle mk_{req}(\underline{\{\, B \mapsto \langle g_i \rangle \,\}}) \rangle \,\}$$

```
sem (Sreq gs _) = MkT [mkt Treq (true,ch)]
    where ch = Choice (upB)
          upB w = nullWorld #> (idB, (Dpgr pgr'))
              where Dpgr pgr = w!idB
                    pgr' = pgr `union` mPriGrp gs
```

3.3.10 PriAlt-Wait

The $wait\langle g_i\rangle$ statement is very similar to the $while$ statement; however, it involves (multiple) different phases. The terminating guarded event (`tslot`) occurs during the res phase, while the continuation guarded event (`cslot`) occurs during the act phase. This is because the continuation phase can continue through multiple micro-cycles (for example, if default guards are present in other branches) in case any of the waiting guards have become available. However, once a (non-default) guard becomes active, the channel communication immediately takes place, which takes a clock step.

$$
\begin{aligned}
\llbracket wait\langle g_i\rangle\rrbracket \quad \widehat{=} \quad \mathsf{fix}\ \mathcal{W} \bullet\ & \{\, \langle mk_{res}(\overline{\neg\ w\langle g_i\rangle})\rangle \,\} \\
& \cup \\
& \overline{w\langle g_i\rangle}\colon^{\{\,\}}_{act} (\llbracket^{+}\langle g_i\rangle\rrbracket \,\{\fatsemi\}\, \mathcal{W})
\end{aligned}
$$

```
sem (Swait gs _) = tcons tslot (MkT ctrc)
    where tslot = mkt Tres (notp (Swaits gs NA),enull)
          MkT ctrc = tmap (($:) cslot) (sem cont)
          cslot = mks Tact (Swaits gs NA,enull)
          cont = sseq [sdelay 1,Sreq gs NA,Swait gs NA]
```

3.3.11 PriAlt-Case

The `prialt` case statement creates a separate guarded event for each possible p_i, where exactly one guard will be executed duing the res phase.

$$
\llbracket a\langle g_i\rangle \blacktriangleright [p_i]\rrbracket \quad \widehat{=} \quad \bigcup_i \{\, \overline{(a\langle g_i\rangle = i)}\colon^{\{\,\}}_{res} \llbracket p_i\rrbracket \,\}
$$

```
sem c@(Scond sel ps _)
    = MkT $ intr 0
      [ branch (n−1) c | n ← [1..(length ps)] ]
    where
```

```
    branch :: Int → SStmt NullAttr → [TATi]
    branch i (Scond sel@(Ssel gs _) ps _)
        = map (asst $:) action
        where asst = mks Tres (sequal sel (sint (i+1)),enull)
              (MkT action) = sem (sseq [act (gs!!i),ps!!i])

sem _ = error "Unknown␣statement␣type"
```

3.3.12 Useful Functions

Merge TrcSets

Apply TAT functions to whole TrcSets, interleaving the results as appropriate. It is used in the semantics for sequential and parallel composition. This function is highly dependent on the interleave function, discussed in Section 6.2.3.

```
mergeTS :: (TATi → TATi → TATi) → [TrcSet] → TrcSet
mergeTS t ts
    = foldr1 op ts
   where
   op (MkT ps) (MkT qs)
     = MkT (map (uncurry t) (ps `interleave` qs))
```

act

The act function gives equivalent statements for a guard:

$$\begin{array}{rcl}
\mathsf{act}() & : & Grd \rightarrow Prog \\
\mathsf{act}(c!e) & \widehat{=} & \mathbf{1} \\
\mathsf{act}(c?v) & \widehat{=} & v := \delta(c) \\
\mathsf{act}(!?) & \widehat{=} & \mathbf{0}
\end{array}$$

```
act :: SGuard NullAttr → SStmt NullAttr
act (Sout c e a) = Sdelay 1 a
```

```
act (Sin c v a)  = Sassign v (delta c) a
act (Sdefault a) = Sdelay 0 a

delta :: Ch → SExpr NullAttr
delta c = Schan c NA
```

3.4 Running and Stepping

3.4.1 Stepping

The step functions takes a $TrcSet$ and a World, and advances all the Traces in the set (discarding the traces that are no longer valid). The valid traces are used to update the World, and the new $TrcSet$ and World are returned a tuple.

$$step : \{\, Trc \,\} \rightarrow World \rightarrow (\{\, Trc \,\}, World)$$

There are two different types of stepping; stepping by microslots (the microslot cycle is $[(sel, req, res)^{+}act])^{*}$, and stepping by full slots (only act transitions) which is equivalent to stepping by full clocksteps.

Both denStep and denMstep are implemented in terms of stepTrcSet:

```
denStep,denMstep :: TrcSet → World → (TrcSet,World)
denMstep  = stepTrcSet tstepMS
denStep   = stepTrcSet tstepSlot
```

Step TrcSet

stepTrcSet is a wrapper around a lower-level step function which handles stepping each individual trace. This function handles combing the outputs of these traces back into a $TrcSet$.

```
stepTrcSet :: (World → TATi → Maybe (TATi,World))
       → TrcSet → World → (TrcSet,World)
stepTrcSet stepF (MkT []) w = (MkT [],w) -- no change
stepTrcSet stepF (MkT ts) w = (ts',w')
    where rs =  catMaybes (map (stepF w) ts)
          w' = (snd.head) rs
          ts' = MkT (map fst rs)
```

Step a trace one slot

tstepSlot steps a trace a single slot. It accomplishes this by continually skipping microslots until a act transition is found, and then returns that result. The function also performs the following functions:

- When a act transition is found, reset the dynamic state $(\Re, \gamma, B)$
- When a $req \rightarrow res$ transition is found, update Re

Much of the functionality in this function comes from the getNextGE function, which selects the next guarded event in a trace and determines its transition type.

```
tstepSlot :: World → TATi → Maybe (TATi,World)
tstepSlot w [] = Just ([],w) — no more traces
tstepSlot w [s]              — handle case where ss = null
    | s == snull = Just ([],w)
    | otherwise  = tstepSlot w [s,snull]
tstepSlot w (s:ss)
    | valid && tt==Tact = Just (t,zeroDynSt w′)
    | valid && tt==Treq = tstepSlot (updateRes  w′) t
    | valid             = tstepSlot w′ t
    | otherwise         = Nothing
    where (ge,rs,tt) = getNextGE s
          (b,p)      = ge
          t          = rs $: ss
          valid      = assert b w
          ch         = apply p w
          w′         = updateW ch w
```

Stepping microslots

tstepMS steps one micro-slot. It is almost identical to tstepSlot, but does not recurse if the transition type is not act. Ideally these two functions could be combined, with tstepSlot calling tstepMS; however, there first must be a way to get the current transition type from the output of tstepMS.

```
tstepMS :: World → TATi → Maybe (TATi,World)
tstepMS w [] = Just ([],w) — stepping after traceDone
tstepMS w [s]              — handle case where ss = null
    | s == snull = Just ([],w)
    | otherwise  = tstepMS w [s,snull]
tstepMS w (s:ss)
```

```
    | valid && tt==Tact = Just (t,zeroDynSt w')
    | valid && tt==Treq = Just (t,updateRes w')
    | valid             = Just (t,w')
    | otherwise         = Nothing
    where (ge,rs,tt) = getNextGE s
          (b,p) = ge
          t = rs $: ss
          valid = assert b w
          w' = updateW ch w
          ch = updateTT tt (apply p w)
```

Get next guarded event

getNextGE is given a slot, and returns the next guarded event, the remaining slot, and the transition type.

```
getNextGE :: Sloti → (GEi,Sloti,TType)
getNextGE (mss,ge)
    | null mss  = (ge,  snull,                     Tact)
    | selp      = (sel, ((genil,req,res):ms,ge),   Tsel)
    | reqp      = (req, ((genil,genil,res):ms,ge),Treq)
    | resp      = (res, (ms,ge),                   Tres)
    | otherwise = getNextGE (ms,ge)
    where [selp,reqp,resp] = map (not.geIsNil) [sel,req,res]
          (m:ms) = mss
          (sel,req,res) = m
```

3.4.2 Running

denRun "prunes" a *TrcSet* by applying an initial world, and removing all traces that are not valid, until there is only one trace left, creating a list of worlds for each clockstep. It is implemented via denStep.

$$run : \{\, Trc \,\} \rightarrow World \rightarrow \langle World \rangle$$

```
denRun :: TrcSet → World → [World]
denRun ts w
    | traceDone ts = []
    | otherwise     = nw : denRun nts nw
   where (nts,nw) = denStep ts w
```

3.4.3 Denotational Semantics State

In order to include the denotational semantics as an availble semantic mode, it must have a state-based representation. The "state" of the denotational semantics is simply the current *TrcSet* and world.

```
type DState = (TrcSet,World)
```

In addition to running and stepping, it must be possible to initialize and disply a state.

```
initDS :: SStmt NullAttr → DState
initDS p = (sem p,mkWorld p) — (MkT [],nullWorld)

fmtDState :: DState → String
fmtDState (t,w)
   = "Time:␣" ++ show (w!idTau)
     — ++ "λnTraces:λn" ++ show t
     ++ "λnWorld:λn" ++ fmtEnv 4 w

fmtTraces :: DState → String
fmtTraces (t,w) = "λnTraces:λn" ++ show t
```

Chapter 4

Handel-C Simulator

4.1 Comparison Module

This module is responsible for comparing different environments, and sets of environments (pruned traces). It is a key component of both the simulator (Chap. 4) and the QuickCheck tests (Chap. 5).

4.1.1 Compare Current Environment

This function compares two environments (of any type), and returns a set of the differing variables ($\emptyset$ if none).

```
compareEnv :: (Ord a,Ord b) ⇒ Env a → Env b → Set Id
compareEnv s1 s2 = dom (n1 `diffMap` n2)
    where n1 = naEnv s1
          n2 = naEnv s2
```

4.1.2 Compare Traces

The compareTraces function compares the pruned results of traces (a sequence of environments). The comparison will result in one of three results:

1. Both terminate at clock cycle n; final state is ...
2. Traces diverge at clock cycle n; differences are ...
3. Traces still match after t steps (possibly infinite?)

In order to accomplish this, t (the maximum number of steps to try) must be an argument. Note that any type of environment is allowed.

```
compareTraces :: (Ord a,Ord b,Show a,Show b) ⇒
                 Int → (String, [Env a]) → (String, [Env b]) → String

compareTraces t (nm1,e1) (nm2,e2)
    | t==n      = msgInfi n (last same)
    | null fail = msgSucc n (last same)
    | otherwise = msgFail n (nm1,nm2) (head fail)
   where n1 = map cEnv e1 -- Env a → Env NA
         n2 = map cEnv e2 -- Env b → Env NA
         ts = zip n1 n2
         (pass,fail) = break (uncurry (/=)) ts
         same = map fst (take t pass)
         n = length same
```

The cEnv calls change the type of the environments to *NullAttr*. Additionally, $\Re$ is removed from the environments; this is because $\Re$ is not part of the externally-visible Handel-C state. In particular, $\Re$ caused problems with testing the **Pri-Def** Law (in Section A.2) because the test for that law produces different initial values for $\Re$.

```
cEnv e = (naEnv ∘ (mremove (iSet idRes))) e
```

4.2 Simulator Changes

The overall design of the simulator has been greatly changed.

The major differences are:

- Support for multiple semantics, via the mode command
- Support for multiple files, via the load2, mode2, and clear2 commands
- An enhanced load command; it is now possible to load a program from a file, `stdin`, or randomly generate one.
- It is possible to list all `.hcp` files in the current directory
- Added a meta-command, time, to time any other command
- Added/improved commands for running the simulation:

mstep steps a program through its semantics, at the micro-step level (if available).

step steps a problem one clockstep. It takes an optional argument, n, designating how many clocksteps to step. If multiple files are present, they are both stepped one clocks step, and their current environments are compared.

run runs a program, step-by-step, until it has terminated (or until a user-defined limit has been reached). If two programs are present, it will run them both, and compare their complete set of results.

4.3 Problems with Original Tool

A few problems were found in the original tool; in particular, the operational semantics for the `prialt wait` had been modified since the tool was written. The implementation of the operational semantics was changed to solve this problem. Some minor changes were made to other modules; for a complete list, see Appendix B.

4.4 Semantic State

This module is a unified state class/datatype for different semantics. It is designed to facilitate adding new semantic "modes"; for example, it should be fairly easy to add a mode for the Handel-C "Compilation Semantics."

4.4.1 Type of semantics

Currently, the only modes supported are the operational and denotational semantics.

```
data SemType = DenSem
             | OpSem
               deriving (Eq, Show, Read)
```

4.4.2 Semantic state class

In order to add a semantic mode to this module, it must implement the SemState class, defined below. A member of this class must implement all the of functions mentioned; for example, to initialize the state from a program, step the state (generating a new state), and run the state (generating a list of environments).

```
class SemState a where
    initS    :: SStmt NullAttr → a
    step     :: a → a
    mstep    :: a → a
    run      :: a → [Env NullAttr]
    stepN    :: Int → a → a
    ssEnv    :: a → Env NullAttr
    putEnv   :: Env NullAttr → a → a
    fmtState :: a → String
    getState :: a → SemType

    mstep = step
    step = stepN 1
```

Denotational SemState

The instantiation of the denotational semantics mode as a SemState

```
instance SemState DState where
    initS    = initDS
    fmtState = fmtDState
    mstep    = uncurry denMstep
    step     = uncurry denStep
    stepN n  = uncurry (denStepN n)
    run      = uncurry denRun
    ssEnv (_,e) = e
    putEnv e (t,_) = (t,e)
    getState _ = DenSem
```

Operational SemState

The instantiation of the operational semantics mode as a SemState (PState)

```
instance SemState PState where
```

```
initS     = initP
fmtState = fmtPState
mstep     = opStep
step      = opStepTick
stepN     = opStepN
run p     = map naEnv (opRun p)
ssEnv (_,_,e)  = naEnv e
putEnv e (p,tt,_) = (p,tt,ttEnv e)
getState _ = OpSem
```

4.4.3 Semantic state abstract implementation

In order to store an arbitrary semantic state, an abstract semantic state data type is created. This data type implements the SemState class itself, and maps the function calls to the appropriate implementation.

This code is based on similar code for supporting multiple types of traces in the simple imperative simulator discussed in Chapter 3. This technique was first attempted using parameterized datatypes (e.g., the simulator state data structure (`PST`) would contain a `SemState a`, and would be called a `PST a`. However, this quickly grew unworkable, notation-wise, as the state currently requires two modes to be kept (which would be `PST a b`), and all function signatures would have also have to include these parameters and associated restrictions. This type of object-oriented solution might be easier to implement in an object-oriented variant such as O'Haskell[1].

The first step is to add the new semantic state to the different possible implementations of the abstract SState:

```
data SState = SemDS DState
            | SemOS PState
```

Next, add mappings between the abstract state and semantic state for each function. Since each implementation of SState must support the SemState class, this is a very simple process.[2].

```
instance SemState (SState) where
```

---

[1] `http://www.cs.chalmers.se/~nordland/ohaskell/`

[2] One that could be easily automated, if this functionality were to be added to the language itself

```
initS ss           = error "initS␣not␣implemented;␣use␣initSS"
fmtState (SemDS s) = fmtState s
fmtState (SemOS s) = fmtState s
mstep    (SemDS s) = SemDS $ mstep s
mstep    (SemOS s) = SemOS $ mstep s
step     (SemDS s) = SemDS $ step s
step     (SemOS s) = SemOS $ step s
stepN n  (SemDS s) = SemDS $ stepN n s
stepN n  (SemOS s) = SemOS $ stepN n s
run      (SemDS s) = run s
run      (SemOS s) = run s
ssEnv    (SemDS s) = ssEnv s
ssEnv    (SemOS s) = ssEnv s
putEnv e (SemDS s) = SemDS $ putEnv e s
putEnv e (SemOS s) = SemOS $ putEnv e s
getState (SemDS s) = getState s
getState (SemOS s) = getState s
```

Finally, the function below must be updated in order to initialize a new SState based on a SemType

```
initSS :: SStmt NullAttr → SemType → SState
initSS ss m
  | m==DenSem = SemDS (initS ss)
  | m==OpSem  = SemOS (initS ss)
```

4.5 Multiple Files

Adding support for multiple files was a non-trivial change. In the original simulator, all file information was stored with the rest of the program state (in a data structure called the PST). If the file data were removed from the PST, all the current commands would need to be rewritten, including all the commands which only dealt with one file. To get around this problem, the original file data was left in the PST, but an additional constructor was added with the file data subset for the second file. The revised data structure is shown in Listing 4.1

In order to make it easy to run a command on one or both files, file-based functions

Listing 4.1: Revised PST

```
data PST
  = PST
    {fname   :: String,   -- file name
     semst   :: SState,   -- current semantic state data
     semmode :: SemType,  -- type of current semantic state (OpSem|DenSem)
      (other file data)
     tmax    :: Int,      -- maximum number of steps to try
     file2   :: PST,      -- alternate file (really a FST)
     dmode   :: Bool      -- flag for single or dual file mode
      (other simulator-wide data)
    }
 | FST
    {fname   :: String,   -- file name
     semst   :: SState,   -- current semantic state
     semmode :: SemType,  -- type of current semantic state (OpSem|DenSem)
      (other file data)
    }
```

had a signature of `PST -> IO PST`, but would only modify data for the primary file. The command would then be wrapped in another meta command, cmdBoth, with signature `(PST -> IO PST) -> PST -> IO PST`. This command takes an actual command, *cmd*, and runs it on the PST. It then checks to see if the simulator is in dual-file mode, and if so, runs *cmd* on `file2 PST`, which is also of type `PST`. cmdBoth then aggregates the results back into a single `IO PST`, and returns it.

Chapter 5

Handel-C QuickCheck Support

5.1 QuickCheck Generators

5.1.1 Random Environments

genRandomWorld takes a set of Ids to initialize, and randomly sets the Ids by calling ival on each one.

```
genWorld :: Ord a ⇒ Set Id → Gen (Env a)
genWorld ids
     = do ws ← smapM ival ids
          return (sreduce (mextend,nullMap) ws)
```

genpRandomWorld extends this by taking an initial program as an argument, and only generating Ids for that program.

```
genpWorld :: Ord a ⇒ [SStmt a] → Gen (Env a)
genpWorld ps
    = do let ids = foldr1 union (map pIds ps)
         genWorld ids
```

ival initializes $\Re$, returns a random positive value for τ, and returns an arbitrary Datum for all other variables.

```
ival :: Id → Gen (Env a)
ival id
     | id == idRes = do return (iMap id (Dres dynZero))
     | id == idTau = do t ← genInt
                        let dt = (Dtime t)
                        return (iMap id dt)
     | otherwise   = do d ← arbitrary
                        return (iMap id d)
```

5.1.2 Random Datum

The Datum implementation of arbitrary currently returns a Dint with a random value.

```
instance Arbitrary (Datum a) where
    arbitrary = liftM Dint arbitrary
```

An alternative generator is available for Datums, where there is a certain chance of returning the undefined value (Dundef). The parameter the ratio between the generating Dints and Dundefs. For example, `genDatum 0.0` results in all Dints, while `genDatum 1.0` results in all Dundef values.

```
genDatum :: Float → Gen (Datum NullAttr)
genDatum r
 = frequency  [(i,genInt'),(u,genUndef)]
   where genInt' = do n ← sized $ λn → choose (0,n)
                      return (Dint n)
         genUndef = return (Dundef "")
         i = 10 − u
         u = round (10.0 ∗ r)
```

5.1.3 Random Programs

This function is an arbitrary generator for programs. It returns any statement in the current mini-Handel-C language, although it is biased toward terminating statements ($\mathbf{0}, \mathbf{1}, x := e$).

In order to ensure that random programs are always finite (the program itself, not its execution), this generator uses techniques from (Claessen & Hughes, 2000) to ensure that programs eventually reach one of the terminating states mentioned above. To accomplish this, the *size* parameter is reduced (by a factor of 10) each time a sub-statement is created via a recursive call to this generator. When *size* reaches zero, the program must output one of the terminating statements, thus preventing any infinite programs.

```
instance Arbitrary (SStmt NullAttr) where
    arbitrary = genSStmt
```

```
genSStmt = sized sstmt'
```

```
sstmt' 0 = oneof [return (sdelay 0),                 -- 0
                  liftM sdelay genInt,               -- 1
                  liftM2 sassign genVar sub_expr]    -- v:=e
    where sub_expr = sexpr' 0
sstmt' n | n>0
    = oneof
      [liftM sdelay genInt,                          -- sdelay n
       liftM2 sassign genVar sub_expr,               -- v:=e
       liftM2 (λs1→ λs2→sseq [s1,s2])
              sub_sstmt sub_sstmt,                   -- p;q
       liftM2 (λp1→ λp2→spar [p1,p2])
              sub_sstmt sub_sstmt,                   -- p||q
       liftM3 sif genBool sub_sstmt sub_sstmt,       -- p<|b|>q
       liftM2 swhile genBool sub_sstmt,              -- b*p
       liftM sprialt genGEs]                         -- prialt
    where sub_expr = sexpr' (n`div`10)
          sub_sstmt = sstmt' (n`div`10)
```

5.1.4 Random Expressions

The arbitrary implementation for expressions returns returns either an integer value, a variable, or an operation $(+, -, *)$ on values. Like the statements, it uses *sized* to avoid generating an infinite expression. Note that most expressions are fairly equivalent; it is unlikely that $34 * v + 7 - d$ will return a different value than 1 or v.

```
instance Arbitrary (SExpr NullAttr) where
    arbitrary = genSExpr

genSExpr = sized sexpr'

sexpr' 0 = liftM sint genInt
sexpr' n | n>0
    = oneof
      [liftM sint genInt,
       liftM3 (λo→ λe1→ λe2→ sapp [o,e1,e2]) op sub_expr' sub_expr',
```

```
    liftM svar genVar]
  where op = liftM svar (oneof (map return ["+","-","*"]))
        sub_expr' = sexpr' (n`div`2)
```

5.1.5 Random Guarded Events

In order to generate a random `prialt` statement, a random list of Guarded Events is needed. This list requires the following properties:

- It should contain a default guard 50% of the time.
  - The list may contain at most one default guard.
  - The default guard must be the last item.
- The list will contain 0 or more input and output guards.
- The list cannot mention any channel more than once.
- The channels must be listed in a global ordering.

Channels

To achieve the properties above, we first generate a random list of channels. The channels are of the format and order:

$$\langle c_0, c_1, \ldots, c_n \rangle$$

We need a random, non-null subsequence this sequence.

This is accomplished by generating a random, non-null list of digits (0–9) via the vectorg[1] function. This list is then sorted, and duplicates are removed. Finally, each digit has the letter ‘c’ prepended.

---

[1] vectorg is an implementation of the vector function described in the QuickCheck manual; however, the actual QuickCheck version of vector does something else.

```
genChannels :: Gen [String]
genChannels = do ixs ← sized (λn → vectorg ((n`div`10)+1) genDigit)
                 let sixs = remDups $ sort ixs
                 let cs = map (λx→'c':show x) sixs
                 return cs

genDigit :: Gen Int
genDigit = choose (0,9)
```

Lists of Guards (non-default)

To create a random list of (non-default) guards, a list of channels is generated, and each channel is mapped to either a random variable, or a random expression.

```
genGuards :: Gen [SGuard NullAttr]
genGuards = do cs ← genChannels
               mapM (λc→ oneof [liftM (sinp c) genVar,
                                liftM (sout c) arbitrary]) cs
```

Guards

A default guarded event simply maps !? to a random expression.

Default Guarded Event

```
genDefGuard :: Gen [(SGuard NullAttr, SStmt NullAttr)]
genDefGuard = do e ← arbitrary
                 return [(sdef, e)]
```

Guarded Events

To finally create a list of random guarded events, the guards are simply paired with random events. The default guards are added in with the following ratios:

50% Just input/output guards

25% Input/output guards followed by a default guard

25% Just a default guard

```
genGEs :: Gen [(SGuard NullAttr, SStmt NullAttr)]
genGEs
   = sized (λn → resize (n`div`10) $
              frequency [(2, ges),
                         (1, liftM2 (++) ges genDefGuard),
                         (1, genDefGuard)])
     where ges = liftM2 zip genGuards genEvents
           genEvents = sequence (repeat arbitrary)
```

Parameterized Guarded Events

Some properties (such as the ones in Appendix A.5) require creation of parameterized `prialts`. This function generates a list of guarded events based on parameters min, max, and d. Guards are of the form c$n\sharp$, where $min \leq n \leq max$, and $\sharp = !, ?$. Default guards may be included if $d = True$.

```
genpGEs :: Int → Int → Bool → Gen [(SGuard NullAttr,SStmt NullAttr)]
genpGEs mn mx d
   = sized $ λn → resize (n`div`10) $
     frequency [(2, ges),
                (dg, liftM2 (++) ges genDefGuard),
                (dg, genDefGuard)]
   where ges = liftM2 zip (genpGuards mn mx) genEvents
         genEvents = sequence (repeat arbitrary)
         dg = if d then 1 else 0
```

Generate a list of guards events based on parameters min and max. Guards are of the form c$n\sharp$, where $min \leq n \leq max$, and $\sharp = \mathsf{oneof}\{\,!, ?\,\}$.

```
genpGuards :: Int → Int → Gen [SGuard NullAttr]
genpGuards mn mx
    = do cs ← genpChannels mn mx
         mapM (λc→ oneof [liftM (sinp c) genVar,
                          liftM (sout c) arbitrary]) cs
```

Generate a list of Channel names based on parameters min and max. Channels are of the form cn, where $min \leq n \leq max$.

```
genpChannels :: Int → Int → Gen [Ch]
genpChannels mn mx
    = do inds ← sized $ λn → vectorg (n`div`10) (choose (mn,mx))
         let sinds = remDups $ sort inds
         let cs = map (λx→'c':show x) sinds
         return cs
```

5.2 Testing Trace Equivalence

5.2.1 Trace Equivalence (Op. Sem.)

Test two statements for equality in the operational semantics. By default, 100 steps are checked before deciding that the traces never diverge.

This function checks that all created programs are “well-formed,” and generates a random environment starting environment.

$$s_1 === s_2 \rightarrow Property$$

```
(===) :: SStmt NullAttr → SStmt NullAttr → Property
s1 === s2
    = wellFormed s1 ==>
      wellFormed s2 ==>
      forAll (genpWorld [s1,s2]) $ λe →
      let types = (e::(Env NullAttr))
```

```
        t1 = opTraces s1 e
        t2 = opTraces s2 e
        teq = tracesEq 100 in
    t1 `teq` t2
```

5.2.2 Trace Equivalence (Den. Sem.)

Test two programs for equality in the denotational semantics. Since the denotational semantics simulator is much more resource-intensive than the operational semantics simulator, only 15 steps are checked before deciding that two traces never diverge.

$$s_1 == *s_2 \rightarrow \mathit{Property}$$

```
(==*) :: SStmt NullAttr → SStmt NullAttr → Property
s1 ==* s2
    = wellFormed s1 ==>
      wellFormed s2 ==>
      forAll (genpWorld [s1,s2]) $ λe →
      let types = (e::(Env NullAttr))
          t1 = denTraces s1 e
          t2 = denTraces s2 e
          teq = tracesEq 15 in
      t1 `teq` t2
```

5.2.3 Semantic Equivalence

Test that program has the same traces in both the operational and denotational semantics. The program is checked to ensure that it is “well-formed”, and a random environment is created. The two semantics are compared for at most 15 steps before deciding they never diverge.

$$\mathit{semEq} : s \rightarrow \mathit{Property}$$

```
semEq :: SStmt NullAttr → Property
```

```
semEq p
    = wellFormed p ⟹
      forAll (genpWorld [p]) $ λe →
      nullp p `trivial`
      let types = e::(Env NullAttr)
          op  = opTraces  p e
          den = denTraces p e
          teq = tracesEq 15 in
      op `teq` den
```

Chapter 6

Results

6.1 Typed Assertion Traces

6.1.1 Concatenating for Microslots

We can now define a form of concatenation for microslots ($\fatsemi$) which merges the last micro-slot of the first sequence (*ante-slot*) with the first micro-slot of the second (*post-slot*), if possible. This is possible when no event in the ante-slot has a type greater than that of an event in the post-slot. We first define an operator ($\boxplus$) taking a pair of micro-slots to a sequence of same:

$$
\begin{array}{rcl}
\_ \boxplus \_ & : & MS^2 \rightarrow MSS \\
(s_1, \mathbf{1}, \mathbf{1}) \boxplus (s_2, q_2, r_2) & \widehat{=} & \langle (s_1 \Diamond s_2, q_2, r_2) \rangle \\
(s_1, q_1, \mathbf{1}) \boxplus (\mathbf{1}, q_2, r_2) & \widehat{=} & \langle (s_1, q_1 \Diamond q_2, r_2) \rangle \\
(s_1, q_1, r_1) \boxplus (\mathbf{1}, \mathbf{1}, r_2) & \widehat{=} & \langle (s_1, q_1, r_1 \Diamond r_2) \rangle \\
m_1 \boxplus m_2 & \widehat{=} & \langle m_1, m_2 \rangle
\end{array}
$$

This equation previously tested for ∇ (genull) instead of $\mathbf{1}$ (genil). However, ∇ only tests to see if the event in a micro-slot is null, and this is *always* true for both sel and res microslots:

$$
sel, res \quad = \quad (Guard, \oslash)
$$

This caused a bug where sel and res microslots would never be recognized as valid, and would be discarded. The function was also reformatted to be more concise, although some boolean values (`r1nil`,`s2nil`) are now evaluated twice. The final version can be seen in Listing 6.1.

Listing 6.1: msglue

```
msglue :: (Prop p, GrdEvt p e) ⇒ MS p e → MS p e → MSS p e
msglue m1@(s1,q1,r1) m2@(s2,q2,r2)
    | q1nil && r1nil = [(s1<.>s2, q2, r2)] — q1,r1 nil
    | r1nil && s2nil = [(s1, q1<.>q2, r2)] — r1,s2 nil
    | s2nil && q2nil = [(s1, q1, r1<.>r2)] — s2,q2 nil
    | otherwise      = [m1,m2]
    where q1nil = geIsNil q1
          r1nil = geIsNil r1
          s2nil = geIsNil s2
          q2nil = geIsNil q2
```

6.1.2 Merging two microslot-sequences in parallel

$$\begin{array}{rcl}
\_ \parallel \_ & : & \mathit{MSS} \times \mathit{MSS} \rightarrow \mathit{MSS} \\
\langle\rangle \parallel \mu_2 & \widehat{=} & \mu_2 \\
\mu_1 \parallel \langle\rangle & \widehat{=} & \mu_1 \\
\mathit{mss}_1 \parallel \mathit{mss}_2 & \widehat{=} & (\mu_1 \parallel \mu_2) \mathbin{;} \langle(m_1 \parallel m_2)\rangle \\
\textbf{where} & & \\
m_1 & = & \mathsf{last}(\mathit{mss}_1) \\
m_2 & = & \mathsf{last}(\mathit{mss}_2) \\
\mu_1 & = & \mathsf{init}(\mathit{mss}_1) \\
\mu_2 & = & \mathsf{init}(\mathit{mss}_2)
\end{array}$$

This function had a bug in which the microslot sequences were merged head-first rather than tail-first. The latter is necessary because of the behavior of the default guard. Consider the program:

$$\langle c!7 \rightarrow \mathbf{0}\rangle \parallel \langle !? \rightarrow \langle c?v \rightarrow \mathbf{0}\rangle\rangle$$

The traces of the first branch of the parallel statement, $\langle c!7 \rightarrow \mathbf{0}\rangle$, are:

$$\begin{array}{ll|l}
[(\mathbf{req}, \mathbf{res}), act] & & 1a \\
[(\mathbf{req}), act] & [(req, res), act] & 2a \\
\vdots & &
\end{array}$$

But for msspar, we are only concerned with the first microslot sequence of each trace (the parts marked in **bold**). Trace $1a$ describes the case where the request for channel c is given, the $Resltn$ shows that this channel is active, and the value 7 is sent along the channel.

The traces for the second half of the expression, $\langle !? \rightarrow \langle c?v \rightarrow \mathbf{0} \rangle \rangle$, are:

$$\begin{array}{lll|l}
[(\mathbf{req}, \mathbf{res}), (\mathbf{req}, \mathbf{res}), act] & & & 1b \\
[(\mathbf{req}, \mathbf{res}), (\mathbf{req}), act] & [(req, res), act] & & 2b \\
[(\mathbf{req}, \mathbf{res}), (\mathbf{req}), act] & [(req), act] & [(req, res), act] & 3b \\
\vdots & & &
\end{array}$$

The first microslot for all these traces is the same; it describes the behavior of the default guard, !?, where it places a request, the $Resltn$ shows that it is active, and its event gets executed in the same clock cycle. Trace $1b$ describes the sequence where the default guard activates, registers the channel request for $c?v$, resolves $c?v$, and activates $c?v$.

The desired behavior of msspar is to merge the first microslot sequence (the parts marked in **bold**) of each branch above. However, the first microslot of the latter branch is always true (since it is a default guard). In fact, all microslots in a sequence other than the last microslot will be caused by default guards.

If a channel is going to activate, the complementary guards must assert their res event at the exact same time. This is due to the fact that in the *Typed Assertion Trace* model, each event is evaluated by run, and if it is not true, the trace is discarded (*pruned*). If one guard asserts the res event before another, the event's guard will evaluate to false. Since a guard can only become active during the *last* microslot of a microslot sequence (since if it activates, it immediately performs a act event), microslots must be merged last-first.

Order does not matter in merging sel events. The static state of the environment (the environment other than $\Re$) cannot change during a microslot sequence, as only act events can change the static environment. It would even be possible to test all sel states in a microslot sequence at once.

The msspar function (shown in Listing 6.2) is implemented somewhat differently from the algorithm above; for efficiency reasons, the lists are only reversed at the beginning and end of the function. Note that msspar_r is identical to the original msspar.

Listing 6.2: msspar

```
msspar :: (Prop p, GrdEvt p e) ⇒ MSS p e → MSS p e → MSS p e
mss1 `msspar` mss2 = reverse (mss1′ `msspar_r` mss2′)
    where [mss1′,mss2′] = map reverse [mss1,mss2]

[] `msspar_r` ms2 = ms2
ms1 `msspar_r` [] = ms1
(m1:ms1) `msspar_r` (m2:ms2) = (m1`mspar`m2):(ms1`msspar_r`ms2)
```

6.2 Traces

In addition to a few minor problems with the definitions of *Typed Assertion Trace*, the actual implementation had to deal with many problems that were not present in the abstract version. For example, *Typed Assertion Trace* contains the idea of trace sets, where the set may be infinite; this is difficult to model in a program. Instead, the trace sets are implemented as a list of traces, where each trace is unique. The order of the list specifies the order in which the traces will be evaluated. Since there may be an infinite number of possible traces, this ordering is very important; the wrong ordering can cause livelock.

6.2.1 While

For example, examine the semantics of the while loop:

```
while (x>0) {
  x = x−1;
}
```

The while loop basically returns two possible traces, either the terminating case (one trace) or the continuing case (infinite traces). Obviously, the terminating case must be placed in the list before the continuing case. Here are the traces, in order, from the while loop above:

	$guard$	$event$	$guard$	$event$	$guard$	$event$	$guard$	$event$
$t_0 =$	$\overline{x > 0}$	$\mathbf{0}$						
$t_1 =$	$x > 0$	$x \mapsto x - 1$	$\overline{x > 0}$	$\mathbf{0}$				
$t_2 =$	$x > 0$	$x \mapsto x - 1$	$x > 0$	$x \mapsto x - 1$	$\overline{x > 0}$	$\mathbf{0}$		
$t_3 =$	$x > 0$	$x \mapsto x - 1$	$x > 0$	$x \mapsto x - 1$	$x > 0$	$x \mapsto x - 1$	$\overline{x > 0}$	$\mathbf{0}$

$\vdots$

Given an initial environment of $x = 2$, run gives us:

	$guard$	$event$	$guard$	$event$	$guard$	$event$	$guard$	$event$
$t_0 =$	$\overline{\mathbf{2 > 0}}$	$False!$						
$t_1 =$	$2 > 0$	$x \mapsto 2 - 1$	$\overline{\mathbf{1 > 0}}$	$False!$				
$t_2 =$	$2 > 0$	$x \mapsto 2 - 1$	$1 > 0$	$x \mapsto 1 - 1$	$\overline{0 > 0}$	$\mathbf{0}$	$True!$	

6.2.2 If

A second example where the order of traces matters is the conditional. A conditional leads to two sets of traces; one if the condition is TRUE, and one where it is FALSE. Imagine a conditional where either possibility leads to a loop; therefore each set of traces is infinite. If one set is tried in its entirety before the other, the latter will never be reached. To avoid this problem, the results must be *shuffled*. This technique is illustrated below:

$$\begin{aligned}
\llbracket p \rrbracket &= \langle trc_1, trc_2, \ldots, trc_n \rangle \\
\llbracket q \rrbracket &= \langle \mathbf{trc_1}, \mathbf{trc_2}, \ldots, \mathbf{trc_n} \rangle \\
\llbracket p \lhd b \rhd q \rrbracket &= \langle trc_1, \mathbf{trc_1}, trc_2, \mathbf{trc_2}, \ldots, trc_n, \mathbf{trc_n} \rangle
\end{aligned}$$

This is implemented via the shuffle function:

shuffle

Shuffle combines two lists by alternating picking between them.

Example: shuffle [1,3,5,7] [2,4,6,8] = [1,2,3,4,5,6,7,8]

```
shuffle :: [a] → [a] → [a]
shuffle x [] = x
shuffle [] x = x
shuffle (x:xs) ys = x : shuffle ys xs
```

6.2.3 Sequential/Parallel Composition

The most difficult problem arises when attempting to combine two trace sets (say, P and Q) via sequential or parallel composition. In either case, the result is a set containing every pairing between the two inputs. Again, if both inputs are infinite, this becomes difficult. Originally, these functions were implemented by pairing every element of Q with $P[0]$, then with $P[1]$, etc. However, if Q is infinite, the pairing of $P[0]$ and Q will never end, and $P[1]$ will never be reached. To avoid this problem, the lists must be combined in a breadth-first manner, as illustrated in Table 6.1.

Table 6.1: Interleave example

	0	1	2	3	4	5	…
A	1	3	6	10	15	21	
B	2	5	9	14	20		
C	4	8	13	19			
D	7	12	18				
E	11	17					
F	16						
⋮							

This results in the two infinite sets being combined as follows:

$$[A, B, C, D, \ldots] \mid\mid\mid [1, 2, 3, 4, \ldots] =$$

$$[(A, 0), (B, 0), (A, 1), (C, 0), (B, 1), (A, 2), (D, 0), (C, 1), (B, 2), (A, 3), \ldots]$$

Interleave

Interleave performs a breadth-first combination of two possibly infinite lists.

```
interleave :: [a] → [b] → [(a,b)]
interleave [] ys = []
interleave xs ys
    = intr 1 ps
    where pair x ys = map (λy→(x,y)) ys
          ps = map (λx→(pair x ys)) xs
```

intr

```
intr :: Int → [[a]] → [a]
intr _ [] = []
intr n xs
    = maplist head fs ++ intr (n+1) ((maplist tail fs) ++ rs)
    where (fs,rs) = splitAt n xs
```

6.2.4 Improving efficiency

While testing various programs in the denotational semantics, some looping programs never managed to loop beyond a certain number of clock-steps. After some investigation, a representative example, labeled `sort-simple` was found:

$$\{\textsc{True}\} * (1 \triangleleft \textsc{False} \triangleright 1)$$

This program could only be stepped 12 or 13 times before becoming overly time (and memory) consuming.

- Used profiling under GHC to figure out why
- Rewrote interleave to be more efficient
- Gained about 10x speed increase
- Den Sem still hangs sometimes

GHC profiling

In order to figure out what was happening, the profiling tools built into GHC[1] were used. This required recompiling the tool with the `-prof` and `-auto-all` flags, and running the command with the `-p` flag:

```
./hcsemtool +RTS -p -RTS
```

The command below[2] was executed in the simulator to create a CPU time/memory allocation profile.

```
comp sort-simple 13
```

Examining the profile output quickly determined that one function, interleave, was dominating the resource usage. This result is shown in Table /reftab:profinterleave in row 1. The command took 69.52 seconds, and an astounding 3.8 GB of memory, to run. The majority (91%) of the time was spent in the interleave function; the only other function that took $> 1\%$ of time or memory was the semE[3] function (which evaluates expressions).

With these results in mind, the interleave function was completely rewritten, and the profiling was re-run. As can be seen in row 2 of Table /reftab:profinterleave, the total time was reduced to 4.6s (a 15x speed increase!). Additionally, the primary bottleneck in the system became semE, although the load was more spread out among functions.

	total time	total alloc	interleave time	interleave alloc	semE time	semE alloc	# functions[a]
run 1	69.52s	3.8 GB	**91.1%**	**94.5%**	3.2%	2.8%	2
run 2	4.6s	229 MB	7.0%	7.9%	**39.1%**	**47.2%**	15

[a] The number of functions which took $>1\%$ of time or memory

Table 6.2: Profiling Interleave Modification

This speed increase, however, is short-lived. For example, the time to run `sort-simple` in the denotational semantics and operational semantics is shown in Table 6.3.

---

[1] `http://www.haskell.org/ghc/`

[2] This command is no longer available, but can be emulated by the commands `load sort-simple ,` `mode DenSem , time s 13`

[3] (The simulator has since switched to the evalE function, but its timing is almost exactly the same as semE)

	10	11	12	13	14	15	16	1,000	10,000
Den. Sem.	1s	2s	4s	8s	16s	31s	61s	…	…
Op. Sem.	57ms	69ms	49ms	67ms	51ms	30ms	63ms	446ms	5s

Table 6.3: Timing operational semantics vs. denotational semantics

The fundamental problem is that, due to the structure of the *Typed Assertion Trace*, there are twice as many traces to investigate for each clock-step. This is because the TRUEcondition in the `if` statement is always tested before the FALSEcondition, and the number of TRUEconditions to test doubles with each clock-step. It may be worth investigating ways to speed up this computation; possibly by switching from a *Typed Assertion Trace*-model to a Tree-based model.

6.3 Testing Semantic Equivalence

One of the primary goals of this work is to test that the operational and denotational semantics are equivalent, both in terms of mathematics and implementation. The semantic equivalence was tested via QuickCheck by using the `semEq` (defined in Section 5.2.3) and the following property:

```
prop_SemEq :: SStmt NullAttr → Property
prop_SemEq p = semEq p
```

6.3.1 Opfail Example

Testing for semantic equivalence generated one failure, shown below:

```
[ a?x → b := 11 ,
  !?  → [ a!22 → c := 33 ,
          !?  → 0 ] ]
```

In this case, the operational semantics returns:

$\{\tau \mapsto 2, b \mapsto 0, c \mapsto \bot\}$

The denotational semantics, on the other hand, returns:

$\{\tau \mapsto 0, b \mapsto \bot, c \mapsto \bot\}$

The latter appears to make more sense, so this initially appeared to be an error in the operational semantics. However, another option is that this syntax is actually illegal, and has no real meaning. To discover what the semantics actually should be, the program was run through the actual Handel-C compiler. The actual Handel-C source code for this program is shown in Listing 6.3.

Listing 6.3: opfail.hcc

```
——— program starts ——————————

set clock = external "P1";
void main(void)
{
    int 8 x,b,c;
        chan int 8 a;  //  Line 6

    par{ b=0; c=0; }
    prialt {           // Line 9
          case a?x :
             b = 11;
                 break;
      default :
             prialt {   // Line 14
                    case a!22 :
                          c = 33;
                          break;
            default:
                           delay;
                           break;
                 };
                 break;
        }
}
———— program ends ———————————
```

The file compiles with 0 errors and warnings; however, during hardware generation gives the errors shown in Fig. 6.3.1.

This test example verifies the the program is not valid Handel-C code. Ideally, we would like to avoid producing similar false-negatives in the future (as we really only want to detect legitimate programs).

```
opfail.hcc Ln 9-23:
  (F0027) Design contains an unbreakable combinational cycle
```

Which expands to

```
opfail.hcc Ln 6, Col 13-14:
  (F0027) Design contains an unbreakable combinational cycle
opfail.hcc Ln 14-21:
  (F0027) Design contains an unbreakable combinational cycle
opfail.hcc Ln 14-21:
  (F0027) Design contains an unbreakable combinational cycle
opfail.hcc Ln 9-23:
  (F0027) Design contains an unbreakable combinational cycle
```

Figure 6.1: `opfail` Handel-C compiler error

There are two possible solutions to avoid similar errors in the future; the first is to test for this condition in the wellFormed invariant of the various compare functions. This could possibly be implemented by checking that there is a clock step present between any two guards with the same channel. The alternative is to have the check actually performed in the various semantics, and return an error condition. The latter solution is probably easier to implement, as it should be simple to generate an error when a channel is re-used in a single clock-step.

6.4 Laws of Handel-C

The Handel-C QuickCheck module was used to test several proposed "Laws of Handel-C." These laws were first encoded as QuickCheck properties using the generators and tests. As noted previously, these laws were tested using the operational semantics because it is much more efficient than the denotational semantics. These laws were then run through QuickCheck, where each law was tested with 1000 different randomly-generated test cases, and any failures were noted.

Overall, 28 proposed laws were tested (some proposed laws were not tested because they are not currently expressible in the "mini-Handel-C" syntax, or because they deal with error conditions). These tests exposed a couple problems with the implementation itself. For example, (give Pri-Def example).

Once these initial "bugs" were fixed, however, all of the laws passed successfully. Three examples of these laws are given below; one simple example, one complex example, and one failing example (a modification of one of the others laws.) The complete "Laws of Handel-C," with results, are given in Appendix A.

6.4.1 Simple Example

This is example showing that the parallel construct is commutative. p_1 and p_2 are random programs; note how closely the equation matches the implementation.

Par-Comm

$$p_1 \,||\, p_2 \quad = \quad p_2 \,||\, p_1$$

```
prop_ParComm p1 p2
    = spar [p1,p2] === spar [p2,p1]
```

6.4.2 Complex Example

This example is much more complex, especially in terms of the Haskell implementation. The laws states that, if a matching pair of input/output guards exists in two parallel branches, the guards following the matching pair can be ignored.

Pri-Trim

$$\begin{gathered}
\langle g_{11} \rightarrow p_{11}, \ldots, g_{1m-1} \rightarrow p_{1m-1}, c!e \rightarrow p_1, \ldots \rangle \\
|| \\
\langle g_{21} \rightarrow p_{21}, \ldots, g_{2n-1} \rightarrow p_{2n-1}, c?v \rightarrow p_2, \ldots \rangle \\
= \\
\langle g_{11} \rightarrow p_{11}, \ldots, g_{1m-1} \rightarrow p_{1m-1}, c!e \rightarrow p_1 \rangle \\
|| \\
\langle g_{21} \rightarrow p_{21}, \ldots, g_{2n-1} \rightarrow p_{2n-1}, c?v \rightarrow p_2 \rangle
\end{gathered}$$

In order to model this law in QuickCheck, it is necessary to produce valid `prialts` which follow a global priority, but always contain a particular channel. This is accomplished by choosing a particular channel (*c5* in this example), and randomly choosing from a list of channels to go before ($c0$ – $c4$) and after ($c6$–$c9, !?$). This is accomplished via the genpGEs function, described in Section 5.1.5 on page 49.

```
prop_PriTrim p1 p2
  = forAll (genpGEs 0 4 0) $ λgs1h →
    forAll (genpGEs 6 9 1) $ λgs1t →
    forAll (genpGEs 0 4 0) $ λgs2h →
    forAll (genpGEs 6 9 1) $ λgs2t →
    forAll (gco c p1)     $ λco →
    forAll (gci c p2)     $ λci →
    spar [sprialt (gs1h++co++gs1t),sprialt (gs2h++ci++gs2t)]
           ===
    spar [sprialt (gs1h++co),sprialt (gs2h++ci)]
  where c = "c5"
        gci c p = do v ← genVar ; return [(sinp c v, p)]
        gco c p = do e ← arbitrary ; return [(sout c e, p)]
```

6.4.3 Failing Example

This is a slight modification of law Comm-Par, which states that two complementary guards in parallel are equal to assignment ($c!e \;||\; c?v \;=\; v \;:=\; e$). However, if placed in parallel with a random processes, this law should no longer be true, since the parallel statement may mention the same guard, and therefore create an error.

Comm-Par2

$$(c!e \;||\; c?v) \;||\; s \quad = \quad v := e \;||\; s$$

```
prop_CommPar2 e s
  = forAll genVar $ λv →
    spar [spar [sprialt [(sout c e,sdelay 0)],
                sprialt [(sinp c v,sdelay 0)]],
```

```
           s]
              === spar [sassign v e,s]
    where c = "c0"
```

This revised law was tested in QuickCheck to see if the error would be found; in this case, one was found after 68 trials (and it is indeed a case where the same channel name, c_0, is run in parallel). This error is not always found on every set of 100 tests, however, it is found over 50% of the time.

```
HCQuickCheck> test prop_CommPar2
Falsifiable, after 68 tests:
(+ (+ x o) 16)

PRIALT
  c0?c:
    IF (== 1 1)
      d_0
    ELSE
      d_0
"m"
{".Res"|→{}{},".tau"|→3,"c"|→12,"m"|→7}
```

Chapter 7

Conclusion

7.1 Objectives fulfilled

This work has demonstrated an implementation of the current proposed denotational semantics for Handel-C, as described in (Butterfield & Woodcock, 2005a). It features a concrete implementation of the *Typed Assertion Trace* traces model, and discusses various issues not present in its abstract form, including difficulties ordering infinite sequences of infinite traces and performance issues in searching through this data structure. Also, a few minor errors in the *Typed Assertion Trace*are fixed.

The existing operational semantics simulator was rewritten to support different semantic "modes," and both the operational semantics and denotational semantics were added using this model. This was accomplished by using a combined class and datatype module in Haskell, similar to a object in object-oriented programming. The operational semantics implementation was updated to include the `swait` statement, a recent change to the operational semantics.

A comparison module is added to the simulator, allowing different programs and semantic modes to be run and stepped in parallel, comparing the environment at each step.

QuickCheck support was added to model properties of Handel-C programs. This includes generators for statements, expressions, starting environments, and guarded expressions. In conjunction with the comparison module, three equality properties are created: testing that a single program has the same traces in both the operational and denotational semantics, that two programs have the same behavior in the operational semantics, and, likewise, testing two programs for equivalence in the denotational semantics.

The first test is used to demonstrate that the two semantics are equivalent; although some discrepancies between the two semantics are found, they all are found to differences in handing malformed programs.

The second and third tests are used to test 28 proposed "Laws of Handel-C." All of the laws passed, although some errors in the implementation of the properties were found and corrected.

7.2 Future work

7.2.1 Simulator

There are many ways in which the simulator can be further extended. Using the support for multiple semantics "modes," additional modes may be added, such as the partially implemented "hardware compilation" semantics, or support for the actual Handel-C semantics (perhaps through a link to the commercial Handel-C simulator).

The performance of the denotational semantics could be looked into; it should be possible to greatly reduce its time complexity by switching to a different trace model, or by rewriting the code to be tail-recursive.

Several minor additions are possible, such as the ability to pretty-print programs to LaTeX output, integrate the QuickCheck tests into the simulator itself, or even adding a proper GUI to the simulator.

7.2.2 Overall project

The goal of this project is to provide a tool for working with the formal semantics of Handel-C. This tool will ideally be used to help in the formal verification (as opposed to testing) of algebraic laws for Handel-C, and that the denotation and operational semantics are in fact the same. These would be used to create a usable system for formal reasoning about Handel-C programs.

There are some aspects of Handel-C which still need to be formalized, such as the type system and external asynchronous interfaces. Once this is done, all the separate aspects must be combined into a unified framework, which will ideally lead to a practical formal methodology so that programs can be formally specified and refined into actual Handel-C code. It will be important to have practical tool support this process as well; possibly based in part on this work.

Appendix A

Laws of Handel-C

A.1 Law Categories

The laws introduced in this section are split among the following categories:

- Laws that definitely hold (labeled using SMALL CAPS FONT).
- Laws which we may choose to admit or omit, depending on how strict or liberal we want our semantics to be (labeled using *Italic Font*! with a trailing exclamation point). Typically we invoke all these laws for a Handel-C program, but might relax them for a specification. This also includes laws which hold for Handel-C, but really shouldn't!
- Plausible laws that either do not hold for Handel-C, or whose status regarding Handel-C is unclear (labeled using Sans-Serif Font? with a trailing question mark).

A.1.1 Testing Procedure

All laws were tested with a modified version of the QuickCheck command-line utility that runs 1,000 tests (with 10,000 tests generated before the "Arguments exhausted after n tests" error). All tests passed, although three tests ran out of arguments, after passing a reasonable number of tests.

A.2 Structural Laws

SEQ-ID

$$\mathbf{0};\ p \quad = \quad p \tag{A.1}$$

$$p \quad = \quad p;\ \mathbf{0} \tag{A.2}$$

```
prop_SeqId_L p = sseq [sdelay 0,p] === p
prop_SeqId_R p = p === sseq [p,sdelay 0]
```

```
*HCLaws> prop_SeqId_L: OK, passed 1000 tests.
*HCLaws> prop_SeqId_R: OK, passed 1000 tests.
```

PAR-ID

$$\mathbf{0} \,||\, p = p \tag{A.3}$$

$$p = p \,||\, \mathbf{0} \tag{A.4}$$

```
prop_ParId_L p
    = spar [sdelay 0,p] === p
prop_ParId_R p
    = p === spar [p,sdelay 0]
```

```
*HCLaws> prop_ParId_L: OK, passed 1000 tests.
*HCLaws> prop_ParId_R: OK, passed 1000 tests.
```

SEQ-ASSOC

$$p_1;\ (p_2;\ p_3) = (p_1;\ p_2);\ p_3 \tag{A.5}$$

```
prop_SeqAssoc p1 p2 p3
    = sseq [p1, sseq [p2,p3]] === sseq [sseq [p1,p2], p3]
```

```
*HCLaws> prop_SeqAssoc: OK, passed 1000 tests.
```

PAR-ASSOC

$$p_1 \,||\, (p_2 \,||\, p_3) = (p_1 \,||\, p_2) \,||\, p_3 \tag{A.6}$$

```
prop_ParAssoc p1 p2 p3
    = spar [p1, spar [p2,p3]] === spar [spar [p1,p2], p3]
```

```
*HCLaws> prop_ParAssoc: OK, passed 1000 tests.
```

PAR-COMM

$$p_1 \mid\mid p_2 \quad = \quad p_2 \mid\mid p_1 \tag{A.7}$$

```
prop_ParComm p1 p2
    = spar [p1,p2] === spar [p2,p1]
```

```
*HCLaws> prop_ParComm: OK, passed 1000 tests.
```

COND-SEQ

$$(p_1 \lhd c \rhd p_2);\ s \quad = \quad (p_1;\ s) \lhd c \rhd (p_2;\ s) \tag{A.8}$$

```
prop_CondSeq p1 p2 s
    = forAll genBool $ λc →
      sseq [sif c p1 p2,s] === sif c (sseq [p1,s]) (sseq [p2,s])
```

```
*HCLaws> prop_CondSeq: OK, passed 1000 tests.
```

CASE-SEQ

$$(e \blacktriangleright [p_1, \ldots, p_n]);\ s \quad = \quad e \blacktriangleright [(p_1;\ s), \ldots, (p_n;\ s)] \tag{A.9}$$

```
prop_CaseSeq s
    = forAll (sized $ λn → vector (n+1)) $ λss →
      forAll (genCase $ length ss) $ λe →
      sseq [scond e ss, s]
              ===
      scond e (map (append s) ss)
    where append x y = sseq [y,x]
```

```
*HCLaws> prop_CaseSeq: Arguments exhausted after 603 tests.
```

PRI-SNGL

$$\langle g \rightarrow p \rangle \quad = \quad \langle g \rightarrow \mathbf{0} \rangle;\ p \tag{A.10}$$

Note, the test below currently only tests input and output guards, not default default guards.

```
prop_PriSngl g p
  = sprialt [(g,p)] === sseq [sprialt [(g,sdelay 0)],p]
```

```
*HCLaws> prop_PriSngl: OK, passed 1000 tests.
```

PRI-DEF

$$\langle !? \rightarrow p \rangle \quad = \quad p \tag{A.11}$$

```
prop_PriDef p
  = sprialt [(sdef,p)] === p
```

```
*HCLaws> prop_PriDef: OK, passed 1000 tests.
```

GRD-SNGL

$$g \quad = \quad \langle g \rightarrow \mathbf{0} \rangle \tag{A.12}$$

Not implemented (in current subset of language, $g \ \widehat{=}\ \langle g \rightarrow \mathbf{0} \rangle$!)

SLF-SNGL

$$!? \quad = \quad \mathbf{0} \tag{A.13}$$

Not implemented (not supported by the current subset of language).

Seq-Zero!

$$\bot;\ p \quad = \quad \bot \tag{A.14}$$

$$\bot \quad = \quad p;\ \bot \tag{A.15}$$

Not implemented.

Par-Zero!

$$\bot \,||\, p \quad = \quad \bot \tag{A.16}$$

$$\bot \quad = \quad p \,||\, \bot \tag{A.17}$$

Not implemented.

A.3 Conditional Laws

COND-TRUE

$$p_1 \lhd \mathbf{true} \rhd p_2 \quad = \quad p_1 \tag{A.18}$$

```
prop_CondTrue p1 p2
    = sif strue p1 p2 === p1
```

```
*HCLaws> prop_CondTrue: OK, passed 1000 tests.
```

COND-FALSE

$$p_1 \lhd \mathbf{false} \rhd p_2 \quad = \quad p_1 \tag{A.19}$$

```
prop_CondFalse p1 p2
    = sif sfalse p1 p2 === p2
```

```
*HCLaws> prop_CondFalse: OK, passed 1000 tests.
```

CASE-SEL

$$(i \blacktriangleright [p_1, \ldots, p_n]);\ s \quad = \quad p_i \tag{A.20}$$

```
prop_CaseSel
    = forAll (sized $ λn → vector (n+1)) $ λss →
      forAll (genCase $ length ss) $ λe@(SInt i _) →
      scond e ss === (ss!!(i−1))
```

```
*HCLaws> prop_CaseSel: Arguments exhausted after 594 tests.
```

WHL-COND

$$b * p \quad = \quad (p;\ b * p) \lhd b \rhd \mathbf{0} \tag{A.21}$$

```
prop_WhlCond p
    = forAll genBool $ λb →
      swhile b p ≡≡ sif b (sseq [p,swhile b p]) (sdelay 0)
```

```
*HCLaws> prop_WhlCond: OK, passed 1000 tests.
```

WHL-TRUE

$$\mathbf{true} * p \quad = \quad p;\, (\,\mathbf{true}\, * p) \tag{A.22}$$

```
prop_WhlTrue p
    = swhile strue p ≡≡ sseq [p,swhile strue p]
```

```
*HCLaws> prop_WhlTrue: OK, passed 1000 tests.
```

WHL-FALSE

$$\mathbf{false} * p \quad = \quad \mathbf{0} \tag{A.23}$$

```
prop_WhlFalse p
    = swhile sfalse p ≡≡ sdelay 0
```

```
*HCLaws> prop_WhlFalse: OK, passed 1000 tests.
```

A.4 Event Laws

DLY-SEQ

$$\delta_m;\, \delta_n \quad = \quad \delta_{m+n} \tag{A.24}$$

```
prop_DlySeq
    = forAll genInt $ λm →
      forAll genInt $ λn →
      sseq [sdelay m,sdelay n] ≡≡ sdelay (m+n)
```

```
*HCLaws> prop_DlySeq: OK, passed 1000 tests.
```

DLY-PAR

$$\delta_n \mid\mid \delta_{n+k} \quad = \quad \delta_{n+k} \tag{A.25}$$

```
prop_DlyPar
    = forAll genInt $ λn →
      forAll genInt $ λk →
      spar [sdelay n,sdelay (n+k)] === sdelay (n+k)
```

```
*HCLaws> prop_DlyPar: OK, passed 1000 tests.
```

DLY-DISTR

$$(\delta_n;\ p_1) \mid\mid (\delta_n;\ p_2) \quad = \quad \delta_n;\ (p_1 \mid\mid p_2) \tag{A.26}$$

```
prop_DlyDistr p1 p2
    = forAll genInt $ λn →
      spar [sseq [sdelay n,p1],
            sseq [sdelay n,p2]]
               === sseq [sdelay n, spar [p1,p2]]
```

```
*HCLaws> prop_DlyDistr: OK, passed 1000 tests.
```

EVT-DLY

$$\mathbf{1} \mid\mid v := e \quad = \quad v := e \tag{A.27}$$

```
prop_EvtDly e
    = forAll genVar $ λv →
      spar [sdelay 1,sassign v e] === sassign v e
```

```
*HCLaws> prop_EvtDly: OK, passed 1000 tests.
```

EVT-PAR

$$\mathbf{v_1 := e_1 \mid\mid v_2 := e_2} \quad = \quad \mathbf{v_1 v_2 := e_1 e_2} \tag{A.28}$$

Not implemented (syntax not support by current subset of language).

EVT-PERM

$$\mathbf{v} := \mathbf{e} \quad = \quad \rho((v) := \rho(\mathbf{e}) \tag{A.29}$$

where ρ is a permutation

Not implemented (syntax not support by current subset of language).

EVT-DISTR

$$\begin{gathered}(v_1 := e_1;\ p_1) \mid\mid (v_2 := e_2;\ p_2) \\ = \\ (v_1 := e_1 \mid\mid v_2 := e_2);\ (p_1 \mid\mid p_2)\end{gathered}$$

```
prop_EvtDistr p1 p2 e1 e2
    = forAll genVar $ λv1 →
      forAll genVar $ λv2 →
      spar [sseq [sassign v1 e1, p1],
            sseq [sassign v2 e2, p2]]
               === sseq [spar [sassign v1 e1,sassign v2 e2],
                         spar [p1,p2]]
```

```
*HCLaws> prop_EvtDistr: OK, passed 1000 tests.
```

Comm-Par?

$$c!e \mid\mid c?v \quad = \quad v := e \tag{A.30}$$

```
prop_CommPar e
    = forAll genVar $ λv →
      spar [sprialt [(sout c e,sdelay 0)],
            sprialt [(sinp c v,sdelay 0)]]
               === sassign v e
          where c = "c0"
```

```
*HCLaws> prop_CommPar: OK, passed 1000 tests.
```

Asg-Seq? Not implemented.

Evt-Detm! Not implemented.

Recv-Par! Not implemented.

A.5 `prialt` Laws

Wr-Trim?

$$c!e \mid\mid \langle g_1 \rightarrow p_1, \ldots, g_{n-1} \rightarrow p_{n-1}, c?v \rightarrow p, \ldots \rangle$$
$$=$$
$$c!e \mid\mid \langle g_1 \rightarrow p_1, \ldots, g_{n-1} \rightarrow p_{n-1}, c?v \rightarrow p \rangle$$

```
prop_WrTrim p
    = forAll (genpGEs 0 4 False) $ λgs1 →
      forAll (genpGEs 6 9 True) $ λgs2 →
      forAll (gco c) $ λco →
      forAll (gci c p) $ λci →
      spar [co,sprialt (gs1++ci++gs2)]
              === spar [co,sprialt (gs1++ci)]
    where c = "c5"
          gco c = do e ← arbitrary
                     return $ sprialt [(sout c e, sdelay 0)]
          gci c p = do v ← genVar
                       return [(sinp c v, p)]
```

```
*HCLaws> prop_WrTrim: OK, passed 1000 tests.
```

Rd-Trim?

$$c?v \mid\mid \langle g_1 \rightarrow p_1, \ldots, g_{n-1} \rightarrow p_{n-1}, c!e \rightarrow p, \ldots \rangle$$
$$=$$
$$c?v \mid\mid \langle g_1 \rightarrow p_1, \ldots, g_{n-1} \rightarrow p_{n-1}, c!e \rightarrow p \rangle$$

```
prop_RdTrim p
```

```
    = forAll (genpGEs 0 4 False) $ λgs1 →
      forAll (genpGEs 6 9 True) $ λgs2 →
      forAll (gco c p) $ λco →
      forAll (gci c)   $ λci →
      spar [ci,sprialt (gs1++co++gs2)]
              === spar [ci,sprialt (gs1++co)]
    where c = "c5"
          gci c = do v ← genVar
                     return $ sprialt [(sinp c v, sdelay 0)]
          gco c p = do e ← arbitrary
                       return [(sout c e, p)]
```

```
*HCLaws> prop_RdTrim: OK, passed 1000 tests.
```

Pri-Trim?

$$\langle g_{11} \rightarrow p_{11}, ldots, g_{1m-1} \rightarrow p_{1m-1}, c!e \rightarrow p_1, \ldots\rangle$$

$$||$$

$$\langle g_{21} \rightarrow p_{21}, ldots, g_{2n-1} \rightarrow p_{2n-1}, c?v \rightarrow p_2, \ldots\rangle$$

$$=$$

$$\langle g_{11} \rightarrow p_{11}, ldots, g_{1m-1} \rightarrow p_{1m-1}, c!e \rightarrow p_1\rangle$$

$$||$$

$$\langle g_{21} \rightarrow p_{21}, ldots, g_{2n-1} \rightarrow p_{2n-1}, c?v \rightarrow p_2\rangle$$

```
prop_PriTrim p1 p2
    = forAll (genpGEs 0 4 False) $ λgs1h →
      forAll (genpGEs 6 9 True) $ λgs1t →
      forAll (genpGEs 0 4 False) $ λgs2h →
      forAll (genpGEs 6 9 True) $ λgs2t →
      forAll (gco c p1)     $ λco →
      forAll (gci c p2)     $ λci →
      spar [sprialt (gs1h++co++gs1t),
            sprialt (gs2h++ci++gs2t)]
              ===
      spar [sprialt (gs1h++co),
```

```
          sprialt (gs2h++ci)]
   where c = "c5"
         gci c p = do v ← genVar
                      return [(sinp c v, p)]
         gco c p = do e ← arbitrary
                      return [(sout c e, p)]
```

```
*HCLaws> prop_PriTrim: OK, passed 1000 tests.
```

Pri-Cycl! Not implemented.

Pri-Schd? Not implemented.

Sgl-Sync?

$$\begin{aligned} &\langle c!e \to p_1\rangle \,||\, \langle c?v \to p_2\rangle \,||\, \{\, \langle g_{ij}\rangle_j \,\}_i \\ &\qquad\qquad\qquad = \qquad\qquad\qquad [c \neq g_{ij}] \\ &(v := e;\ (p_1 \,||\, p_2)) \,||\, \langle\{\, g_{ij} \to p_{ij}\,\}_j\rangle_i \end{aligned}$$

```
prop_SglSync p1 p2 e
   = forAll genVar $ λv →
     forAll (sized $ λn → vectorg ((n `div` 10)+1) genGEs) $ λges →
     let c = "c5"
         palts = map genPA ges
         genPA ges = if (null ges') then sdelay 0
                                    else sprialt ges'
            where ges' = filter (lacksCh c) ges in
         spar ([sprialt [(sout c e,p1)],
                sprialt [(sinp c v,p2)]] ++ palts)
                  ===
         spar ([sseq [sassign v e,spar [p1,p2]]] ++ palts)

lacksCh :: Ch → (SGuard NullAttr,SStmt NullAttr) → Bool
lacksCh c (g,e) = cOf g /= c
                  && and (mapSStmt (lacksChS c) e)

lacksChS :: Ch → SStmt NullAttr → Bool
```

```
lacksChS c (Sprialt gss _) = and (map (lacksCh c) gss)
lacksChS c _ = True
```

```
*HCLaws> prop_SglSync: Arguments exhausted after 966 tests.
```

Appendix B

HCSemTool Change Log

Changes to old simulator files (in addition to HCSemToolMAIN, which has been greatly revised).

B.1 HCAbsSyn

1. Extended SExpr with constructor SBool Bool a, which is mostly used by the Denotational Semantics.
2. Added new shorthand expressions for boolean types and default guards:
    - strue, sfalse, sand, sor, snot, sequal, sgt, slt
    - sdef = Sdefault NA
3. Added two functions, which set all attributes to NullAttr
    - clearSAttr :: SStmt a → SStmt NullAttr
    - clearGAttr :: SGuard a → SGuard NullAttr

B.2 HCOpSem

1. Removed TType datatype (since they are also used by Den. Sem.)
2. Added support for "wait" to OpSem:
    - Added new $res \rightarrow req$ transition to enabledT function
    - Added Swait case for fs
    - Added Swait case for tts

- Added Swait case for typeAttr

3. Added support for booleans to if / while

4. Added support functions for running the operational semantics:

   - opTraces :: SStmt a → Env b → [Env TType]
   - opRun :: PState → [Env TType]
   - opStepTick :: PState → PState
   - opStepN :: Int → PState → PState
   - isStop :: PState → Bool
   - getTau :: PState→ Int

5. Added functions for replacing attributes with one of type TType:

   - ttEnv :: Env a → Env TType
   - ttDatum :: Datum a → Datum TType

B.3 HCState

1. Renamed (#) to (#>) (collision with GHC extensions syntax)

2. Moved TType datatype, from HCOpSem (since it is shared with Den. Sem.)

3. Added new TType : Dttype TType, for storing current TType in Den Sem. This was never fully implemented

4. Added support for Dbool to evalE

5. Changed plds to only return Vars, not operators or channels

6. Added following functions to nullify attributes (so that they can be compared to the DenSem) Only environment version is public; the rest are support functions.

   - naEnv :: Ord a => Env a → Env NullAttr

- naDatum :: Ord a => Datum a → Datum NullAttr
- naPgr :: Ord a => PriGrp a → PriGrp NullAttr
- naPAlt :: PriAlt a → PriAlt NullAttr

B.4 TypedAssertionTraces

1. Added function: Typed Event to Trace (as a shorthand).
    - mkt :: GrdEvt p e => TType → GE p e → Trc p e
2. Removed “otherwise’ statement in gemerge.
3. Modified msglue to test of genil instead of genull. Simplified function. This is described in more detail in Section 6.1.1.
4. Merging two microslot-sequences in parallel (msspar) fix; changed to merge last-first, instead of head-first. This is described in more detail in Section 6.1.2.
5. Updated all QuickCheck tests.

References

Butterfield, A. (2001a, December). *Denotational semantics for `prialt`-free Handel-C* (Tech. Rep. No. TCD-CS-2001-53). Trinity College Dublin.

Butterfield, A. (2001b, December). *Interpretive semantics for `prialt`-free Handel-C* (Tech. Rep. No. TCD-CS-2001-54). Trinity College Dublin.

Butterfield, A. (2005). *Handel-C Semantic Tool Manual.*

Butterfield, A., & Woodcock, J. (2002a). Semantic domains for Handel-C. In N. Madden & A. Seda (Eds.), *Mathematical foundations for computer science and information technology (mfcsit 2003)* (Vol. 74).

Butterfield, A., & Woodcock, J. (2002b). Semantics of `prialt` in Handel-C. In J. Pascoe, P. Welch, R. Loader, & V. Sunderam (Eds.), *Communicating process architectures—2002.* IOS Press.

Butterfield, A., & Woodcock, J. (2005a). *Denotational semantics of Handel-C cores.*

Butterfield, A., & Woodcock, J. (2005b). `prialt` in Handel-C: an operational semantics. *International Journal on Software Tools for Technology Transfer (STTT)*, *7*(3), 248–267.

Celoxica Ltd. (2002, August). *Handel-C language overview.*

Celoxica Ltd. (2004). *Handel-C language reference manual, v. 3.0.*

Claessen, K., & Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (pp. 268–279). New York, NY, USA: ACM Press.

Haskell 98 language and libraries: The revised report. (2002, December).

Hoare, C. A. R. (1985). *Communicating sequential processes.* Prentice-Hall.