Specification-Enhanced Execution

by

Jean Yang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 7, 2010

Certified by .....
Armando Solar-Lezama
Assistant Professor
Thesis Supervisor

Accepted by .....
Terry Orlando
Chairman, Department Committee on Graduate Students

Specification-Enhanced Execution

by

Jean Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 7, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Our goal is to provide a framework that allows the programmer to easily shift responsibility for certain aspects of the program execution to the runtime system. We present *specification-enhanced execution*, a programming and execution model that allows the programmer to describe certain aspects of program execution using high-level specifications that the runtime is responsible for executing. With our approach, the programmer provides an implementation that covers certain aspects of program behavior and a set of specifications that cover other aspects of program behavior. We propose a runtime system that uses concolic (combined concrete and symbolic) execution to simultaneously execute all aspects of the program. We describe LogLog, a language we have designed for using this programming and runtime model. We present a case study applying this programing model to real-word data processing programs and demonstrate the feasibility of both the programming and runtime models.

Thesis Supervisor: Armando Solar-Lezama
Title: Assistant Professor

Acknowledgements

I would like to thank my supervisor, Professor Armando Solar-Lezama for his inspiration, dedication, and enthusiasm. Professors Saman Amarasinghe and Martin Rinard have also shaped my education education during these last two years. My fellow CSAIL graduate students Eunsuk Kang, Sasa Misailovic, Aleksandar Milicevic, Joseph P. Near, Rishabh Singh, and Kuat Yessenov have never failed to provide inspiring discussion and helpful feedback. Finally, I would like to thank my parents and my sister Victoria for reminding me about what is important in life.

Contents

1 The Runtime Assistant **9**

1.1 Contributions 10

2 Enhancing Data Processing Programs **11**

2.1 Imputation in data processing 11

2.2 Specification-enhanced execution in LogLog 12

2.3 Executing with specifications 13

3 Specifying and Executing Constraints on Data **16**

3.1 λ-LogLog by example 16

3.1.1 Application and concretization with symbolic values 17

3.1.2 Evaluating concrete expressions 18

3.1.3 Symbolic abstractions vs. function abstractions 19

3.2 All together now 20

3.2.1 Combined concrete and symbolic evaluation 20

3.2.2 Simple typing to ensure normalizing expressions 24

3.2.3 Constraint propagation correctness 24

4 LogLog language design **25**

4.1 Abstract syntax 25

4.1.1 Concrete and symbolic data 27

4.1.2 Types and type-level constraints 27

4.2 Introducing and eliminating constraints 27

4.2.1 Looking up type-level constraints 27
4.2.2 Concretizing the final result 28

5 Implementation **29**
5.1 Bounding space 29
5.2 High-level overview 30
5.3 Interpreter architecture 30
5.3.1 Frontend 30
5.3.2 Backend 30
5.3.3 Solver and interface 31
5.3.4 Frontend interface to backend 32
5.4 Algorithms 32
5.4.1 Frontend 32
5.4.2 Backend 34
5.5 Experience notes 35
5.5.1 Performance wins 36
5.5.2 Ease of implementation vs. runtime performance 37

6 Case Study: Processing Real Census Data **38**
6.1 Current Population Survey and imputation 39
6.1.1 Experimental setup 40
6.1.2 Results 42
6.2 Comparison with other languages 45
6.2.1 SQL 45
6.2.2 Python 46

7 Related work **49**
7.1 Aspect-oriented programming 49
7.2 Programming with specifications 50
7.2.1 Executing specifications 50
7.2.2 Constraint-based programming 50

7.3 Program analysis and repair 50
7.4 Data processing 51
7.4.1 Parsing 51
7.4.2 Constraint databases 51
7.4.3 Data cleaning 51

8 Conclusions and Future Work **52**

8.1 Future work 52

List of Figures

2-1 Census data with all relevant information present. 13

3-1 Evaluation. 22

4-1 Partial LogLog abstract syntax for expressions. Primitives p consist of integers and booleans. Variables can either be identifiers n or field accesses. 26

6-1 Relevant code for imputation tests. 41

6-2 Imputation code in Python. 48

List of Tables

2.1 Census data missing relevant marital status and spouse items. 14

2.2 Census data missing marital status, spouse, and age items. 14

3.1 λ-LogLog syntax. 21

6.1 Imputation rates for different CPS fields by percent [26]. *Household edits* involve hot deck imputation between households. *Demographic edits* involve employing relational, longitudinal, and hot deck imputation based on demographic variables such as marital status, parents, and spouse. 39

6.2 View of the March 2009 CPS data showing the following columns for a subset of entries from household 26974: household ID, line number, age, marital status, spouse line number, marital status imputation flag (MI), and spouse imputation flag (SI). 40

6.3 Times (in seconds) for 1) deriving constraints on the input with incomplete values (Cnsts.), 2) solving the Yices constraints (Yices), and 3) running the interpreter on the entire program (Total). The time for deriving and solving constraints makes up a small fraction of the total running time. We show the total number of records on the left, along with the total number of missing records in the data set combining all missing items. 43

6.4 While constraint evaluation introduces an expected factor of two overhead, symbolic evaluation does not introduce signifcant overheads. . . 44

6.5 Python running times for comparable examples. 47

Chapter 1

The Runtime Assistant

The more the language runtime handles tedious program aspects, the easier it is to write correct programs. Consider garbage collection, which relieves programmers of the tedious and error-prone tasks of memory allocation and deallocation. Software companies have reported productivity gains from switching to garbage-collected languages [28].

While the memory management problem has been solved with clever algorithms and heroic implementations, there are many messy aspects of programming for which there is not yet runtime support. It would be ideal to have a general infrastructure for delegating responsibility to the runtime to handle such program aspects.

Our goal is to develop a general runtime system that can be programmed using high-level declarative specifications. Rather than relying on the programmer to provide implementation details for all parts of the program, we envision having the runtime serve as an assistant that takes responsibility for a set of execution details during execution, alongside the core program. In this framework, the programmer has the flexibility to decide how much to delegate to the runtime.

For implementing flexible runtime assistants we introduce *specification-enhanced execution*, a novel programming and execution model that allows the programmer to transfer responsibility from the core program to the runtime system by introducing *symbolic values* and providing specifications that apply to symbolic expressions. The runtime propagates constraints corresponding to symbolic values in order to derive

outputs consistent with the specifications. The execution model relies on the existence of a *constraint-resolution oracle* for making symbolic values concrete.

To support specification-enhanced execution we introduce the LogLog language and an interpreter implementation that makes use of an SMT (Satisfiability Modulo Theories) solver. We demonstrate the usability of LogLog in the data processing domain, where we can make the runtime responsible for handling missing input values. We reconstruct census data processing examples as documented by the Census Bureau and show that we can process almost 400,000 data entries with thousands of missing in seconds with negligible constraint-solving overhead. This case study suggests that for certain programming tasks that may require considerable programming effort, it is worthwhile to pursue the specification-enhanced execution approach.

1.1 Contributions

This thesis makes the following contributions.

- We present *specification-enhanced execution*, a programming and runtime model that provides the flexibility for transferring responsibility of selected program aspects to the runtime.
- We formally describe specification-enhanced execution and present LogLog, a novel programing language.
- We describe an efficient interpreter implementation for LogLog and demonstrate its performance, scalability, and usability on benchmark programs that processing real census data.

In this thesis, we present specification-enhanced execution in the context of data processing programs, formally describe our programming and execution model, describe the LogLog language and its implementation, and describe a case study in census data processing.

Chapter 2

Enhancing Data Processing Programs

Many programs that process real-world data have simple computational goals but become quite complex due to the need to handle missing inputs. These missing inputs can come from phenomena such as sensor malfunction and survey non-response. We show how to use *specification-enhanced execution* to delegate imputation as a task for the runtime system to perform alongside a program operating on well-formed data. In this chapter, we introduce a running example related to processing census data.

2.1 Imputation in data processing

While it is simple to produce standard census summaries such as total population count from idealized census data, analyzing *real* census data is much more complex due to the significant rate of non-response and response ambiguity and the importance of handling the incomplete data without biasing the results [3]. There are many documented methods for handling this: for instance, Franconi *et al.* write that to account for the nontrivial number of people who fail to declare their marital status, we can use the assumption that marriage relationships are symmetric [10]. This process of assigning values is called *imputation*; imputation methods have a significant effect the outcome of data analysis [3] but are often tedious to implement.

2.2 Specification-enhanced execution in LogLog

We show how to delegate imputation tasks to the runtime. The LogLog runtime processes symbolic expressions alongside concrete ones by propagating constraints that come from specifications associated with type declarations. We can make the runtime responsible for imputation by representing missing values as symbolic variables.

To compute the average age of people who are married, we can write a LogLog program that loads a list of record based on the census_data record type, filters the entries, and takes the average over the age fields:

```
rtype census_data { name     : uid
                  ; age      : integer
                  ; married  : boolean
                  ; spouse   : uid
                  ; mother   : uid
                  ; father   : uid
                  ; children : uid list }

census = load census.txt as (census_data list)
married = filter census by married;
average = avg(married.age);
```

This is the core program that dictates the control flow structure of the execution.

To specify an imputation strategy for census_data records, the programmer can declare a subtype of census_data. Below we show the declaration of the census_data_imputed subtype, which says that for any record r of type census_data_imputed, if the spouse field of r is valid, then the married field should be true:

```
type census_data_imputed = census_data
  with (r) { r.spouse >= 0 implies r.married }
```

We can also relate the entries we have read based on the spouse fields by defining the following list type: .

```
type census_rel_impute_m = census_data_imputed list
  with (rel) {
    forall r, r' in rel : (r.spouse == r'.name) implies
```

Name	Age	...	Married	Spouse	...	Mother
0	28		**true**	3		1
1	48		**false**	–		2
2	70		**true**	11		4
3	34		**true**	0		11
4	88		**true**	39		–

Figure 2-1: Census data with all relevant information present.

```
    ((r.name == r'.spouse) and (r.married == r'.married))
}
```

For all relations rel of type census_rel_impute_m, for any two records r and r' in the relation, there is a symmetric relationship between the spouse fields and the married fields are equal.

We can easily extend the specification to impute ages based on the age of parents and children, either by adding another constraint through conjunction or by subtyping census_rel_impute_m, as we show below:

```
type census_rel_impute_ma = census_rel_impute_m
  with (rel) {
    forall r, r' in rel :
      ((r.mother == r'.name) or (r.father == r'.name)) implies
        r.age < r'.age - 12
}
```

The LogLog runtime will use and propagate these constraints when handling missing values.

2.3 Executing with specifications

An execution over well-formed data simply sums over all the ages and divide by the total. From the data in Table 2-1, the LogLog interpreter would return 55.

An execution over data with missing values makes the missing values symbolic, performs combined concrete and symbolic execution, and makes the symbolic result

Name	Age	...	Married	Spouse	...	Mother
0	28		**true**	3		1
1	48		**false**	-1		2
2	70		**true**	11		4
3	34		–	–		11
4	88		**true**	39		–

Table 2.1: Census data missing relevant marital status and spouse items.

Name	Age	...	Married	Spouse	...	Mother
0	28		**true**	3		1
1	–		**false**	–		2
2	–		**true**	11		4
3	34		–	–		11
4	88		**true**	39		–

Table 2.2: Census data missing marital status, spouse, and age items.

concrete based on constraints derived from types. Below we show the execution of the LogLog interpreter on the data in Table 2.1, in which entry 3 is missing the "Married" and "Spouse" fields.

$$\text{sum}_0 = 28$$

$$\text{sum}_1 = 28$$

$$\text{sum}_2 = 98$$

$$sum_3 = \textbf{if } rel[3].married \textbf{ then } 132 \textbf{ else } 98$$

$$sum_4 = sum_3 + 88$$

The runtime evaluates the married field to the symbolic variable $rel[3].married$ and the sum becomes a symbolic expression. To make a symbolic value concrete, the runtime system will derive a value consistent with the type-level specification. In the case of this example, the constraints will determine that $rel[3].married$ is true.

The runtime system propagates both missing age and marital status information, as we show with respect to the data in Table 2.2:

$$\begin{aligned}
\text{sum}_0 &= 28 \\
\text{sum}_1 &= 28 \\
sum_2 &= 28 + rel[2].age \\
sum_3 &= \textbf{if } rel[3].married \textbf{ then } sum_2 + 34 \textbf{ else } sum_2 \\
sum_4 &= sum_3 + 88
\end{aligned}$$

The runtime system determines rel[3].married is **true** and there should be following bounds on rel[2].age:

$$\begin{aligned}
28 + 12 = 40 < rel[1].age, \\
rel[1].age + 12 < rel[2].age, \\
rel[2].age + 12 < 88.
\end{aligned}$$

This allows us to determine

$$43 < rel[2].age < 88,$$

which allows us to derive

$$48.25 < average < 59.5.$$

Note that the results of such computations retain uncertainty until a concrete value is required.

Chapter 3

Specifying and Executing Constraints on Data

The underlying execution is quite simple: we have a language with two modalities, one for concrete execution and one for handling symbolic values and associated constraints. The key feature is the concept of *symbolic* variables, which designate values that are the responsibility of the runtime. The execution model evaluates expressions containing symbolic values symbolically and associates the resulting symbolic expressions with constraints that are used to make the result concrete. We show how to perform simultaneous concrete and symbolic evaluation, constraint propagation, and concretizing values.

3.1 λ-LogLog by example

We describe the exeuction model using λ-LogLog, the λ-calculus extended with two constructs, *symbolic abstraction* and a *constrained term.* A symbolic abstraction $\sigma y.t$ introduces a *scoped symbolic variable* y in the body of the term t. A *constrained term* $t|_\gamma$ associates the symbolic variables in t with the constraint γ; this constraint is used in making a symbolic expression concrete. For instance, consider the expression $\sigma y.(y|_{y=2y})$. This is a symbolic abstraction over the variable y containing the constrained term $y|_{y=2y}$. This term indicates that the variable y is bound to the

constraint that $y = 2y$. Since this is the only information we have about y, to fully evaluate the σ-term we can resolve the result $y = 0$.

Let us further examine how we can evaluate expressions in this calculus. Recall the following standard λ-calculus functions for pairs (x, y):

mkpair: $\lambda x.\lambda y.\lambda f.(f\ x\ y)$

fst: $\lambda p.p\ (\lambda x.\lambda y.x)$

snd: $\lambda p.p\ (\lambda x.\lambda y.y)$

If we wanted to constrain the two elements of the pairs to be equal, we can write the following pair constructor that makes use of a *constrained term*:

$$\text{mkeqpair} = \lambda x.\lambda y.\lambda f.((f\ x\ y)|_{x=y})$$

Here we have a constructor that takes pair arguments x and y and returns a pair $\lambda f.((f\ x\ y)|_{x=y})$ that constraints x and y to be equal. This constraint applies if either of the arguments are symbolic.

3.1.1 Application and concretization with symbolic values

Consider the case when we construct a pair involving a symbolic value:

$$\begin{aligned}\sigma z.(\text{mkeqpair } 1\ z) &\rightarrow \sigma z.((\lambda y.\lambda f.f\ z\ y|_{z=y})\ 1)\\ &\rightarrow \sigma z.(\lambda f.f\ z\ 1|_{z=1})\\ &\rightarrow \lambda f.(f\ 1\ 1|_{1=1})\end{aligned}$$

We first evaluate the body of the σ expression, getting the constrained symbolic pair $[z, 1]|_{z=1}$. The evaluation of the concretization of σ-bound variable z yields $[1, 1]|_{1=1}$.

Now consider the following evaluation where we have a symbolic variable involved

in a conditional.

$$\begin{aligned}
&\sigma z.(\text{let } x = (\text{mkeqpair } 1\ z) \text{ in } (\text{if } (\text{snd } x = 1) \text{ then } 1 \text{ else } 0)) \\
\rightarrow\ &\sigma z.(\text{if } (z|_{z=1} = 1) \text{ then } 1 \text{ else } 0) \\
\rightarrow\ &\sigma z.((\text{if } (z = 1) \text{ then } 1 \text{ else } 0)|_{z=1}) \\
\rightarrow\ &1
\end{aligned}$$

We evaluate the body of the σ-expression by evaluating the conditional and then concretizing the variable z based on the constraint $z = 1$. When we concretize, we get $z = 1$, so the expression evaluates to 1.

Now suppose instead of returning 0 or 1, we want to increment some variable which happens to be symbolic.

$$\begin{aligned}
&\sigma z.\sigma v.(\lambda s.(\text{let } x = (\text{mkeqpair } 1\ z) \text{ in } (\text{if } (\text{snd } x = 1) \text{ then } s + 1 \text{ else } s))\ (v|_{v \geq 10})) \\
\rightarrow\ &\sigma z.\sigma v.(\text{if } (z|_{z=1} = 1) \text{ then } (v|_{v \geq 10}) + 1 \text{ else } (v|_{v \geq 10})) \\
\rightarrow\ &\sigma z.\sigma v.((\text{if } (z = 1) \text{ then } v + 1 \text{ else } v)|_{(z=1) \wedge (v \geq 10)}) \\
\rightarrow\ &\sigma z.((\text{if } (z = 1) \text{ then } 11 \text{ else } 10)|_{(z=1) \wedge (10 \geq 10)}) \\
\rightarrow\ &11
\end{aligned}$$

We evaluate the expression, simplifying the concrete portions as much as possible and pushing the constraints to the outermost term. We first concretize $v = 10$ according to our constraints, leaving the constraint $z = 1$ on the term because it still contains free symbolic variables. We then concretize z and get a final expression of 11.

3.1.2 Evaluating concrete expressions

The system only applies constraints to symbolic expressions: if a constraint has no symbolic variables, then the system ignores it. Concrete pair creation would cause

the constraint variables to be substituted as follows:

$$\begin{aligned}((\lambda x.\lambda y.\lambda f.(f\ x\ y|_{x=y}))\ 1)\ 0 &\rightarrow (\lambda y.\lambda f.(f\ 0\ y|_{0=y}))\ 1 \\ &\rightarrow \lambda f.(f\ 0\ 1|_{0=1})\end{aligned}$$

Though we clearly have an unsatisfiable constraint here, we have no symbolic variables in the expression, so the constraints will not matter during evaluation. Applying fst will result in the following evaluation:

$$\begin{aligned}(\lambda p.p\ (\lambda x.\lambda y.x))\ (\lambda f.(f\ 0\ 1|_{0=1})) &\rightarrow (\lambda f.(f\ 0\ 1|_{0=1}))\ (\lambda x.\lambda y.x) \\ &\rightarrow ((\lambda x.\lambda y.x)\ 0\ 1)|_{0=1} \\ &\rightarrow ((\lambda y.1)\ 0)|_{0=1} \\ &\rightarrow 1|_{0=1} \\ &\rightarrow 1\end{aligned}$$

Since the constant 1 is fully concrete, it is not under the jurisdiction of the constraints and so we can strip the constraint to produce a value.

3.1.3 Symbolic abstractions vs. function abstractions

According to our evaluation rules, we only evaluate a function abstraction when the function is applied. On the other hand, we evaluate σ-abstractions eagerly, by first evaluating the body and then concretizing the σ-bound variables.

The way to delay the evaluation of a σ-abstraction is to place it inside a λ-abstraction:

$$\lambda f.\sigma y.f\ y.$$

The rules describe the following evaluation:

$$\begin{aligned}(\lambda f.\sigma y.f\ y)\ (\lambda x.x|_{x\geq 0}) &\rightarrow \sigma y.((\lambda x.x|_{x\geq 0})\ y)\\ &\rightarrow \sigma y.(y|_{y\geq 0})\\ &\rightarrow 0.\end{aligned}$$

The evaluation does not concretize the expression until the application of f has been fully evaluated.

We can express the construct "carry out the rest of the computation and then concretize y" if we have the rest of the computation expresses as some *continuation* k. We can write the expression

$$(\lambda f.\sigma y.f\ y)\ k,$$

the evaluation of which would yield $f\ y$ before concretizing y. Given a program written in continuation-passing style, at any point we can introduce σ-abstracted variables to be concretized *after* finishing the rest of the computation.

3.2 All together now

We show the abstract syntax of λ-LogLog in Table 3.1. We only deviate from the standard λ-calculus with the *symbolic abstraction* and *constrained term*. Note that values only have constraints if they are under a λ-abstraction.

3.2.1 Combined concrete and symbolic evaluation

In 3-1 we describe the rules for our mixed concrete and symbolic evaluation. The rules assume all variables have unique names. Note that the $\mapsto$ substitutes variable bindings in constraints as well as in term bodies.

The interesting rules are CONCRETIZE, which evaluates an expression and then assigns a concrete value to the result, the rules OPSYM and CONDSYM, which

t	$::=$		*terms:*
		x	*variable*
		$t\ t$	*application*
		$t\ op\ t$	*arithmetic/boolean operation*
		if t then t else t	*conditional*
		v	*value*
		$\sigma y.t$	***symbolic abstraction***
		$t\|_\gamma$	***constrained term***
v	$::=$		*values:*
		$\lambda x.t$	*abstraction value*
		i	*integer*
		b	*boolean*
γ	$::=$		*constraint functions:*
		t	*term*

Table 3.1: λ-LogLog syntax.

Symbolic abstraction:

$$\frac{t \to t'|_\gamma \quad \gamma \to \gamma' \quad SAT((y = c) \wedge \gamma') \quad [y \mapsto c]t'|_{[y \mapsto c]\gamma'} \to t_f}{\sigma y.t \to t_f} \text{ (CONCRETIZE)}$$

Function application:

$$\frac{t_1 \to t_1' \quad t_1'\ t_2 \to t_r'}{t_1\ t_2 \to t_r'} \text{ (APP)} \qquad \frac{t_2 \to t_2' \quad [x \mapsto t_2]t_1 \to t'}{(\lambda x.t_1)\ t_2 \to t'} \text{ (APPABS)}$$

$$\frac{t_c \to t_c'|_{\gamma_c} \quad t_a \to t_a' \quad [x \mapsto t_a']t_1 \to t_1'|_{\gamma_1} \quad [y \mapsto t_a']t_2 \to t_2'|_{\gamma_2}}{(\text{if } t_c \text{ then } (\lambda x.t_1) \text{ else } (\lambda y.t_2))\ t_a \to (\text{if } t_c' \text{ then } t_1' \text{ else } t_2')|_{\gamma_c ? \gamma_1 : \gamma_2}} \text{ (APPSYM)}$$

Arithmetic and boolean operations:

$$\frac{t_1 \to v_1 \quad t_2 \to v_2 \quad v_1\ op\ v_2 = v_r}{t_1\ op\ t_2 \to v_r} \text{ (OPABS)} \qquad \frac{t_1 \to t_1'|_{\gamma_1} \quad t_2 \to t_2'|_{\gamma_2}}{t_1\ op\ t_2 \to (t_1'\ op\ t_2')|_{\gamma_1 \wedge \gamma_2}} \text{ (OPSYM)}$$

Conditionals:

$$\frac{t_c \to \text{true}|_\gamma \quad t_t|_\gamma \to t_t'}{\text{if } t_c \text{ then } t_t \text{ else } t_f \to t_t'} \text{ (CONDABS} - \text{TRUE)}$$

$$\frac{t_c \to t_c'|_{\gamma_c} \quad t_t \to t_t'|_{\gamma_t} \quad t_f \to t_f'|_{\gamma_f}}{\text{if } t_c \text{ then } t_t \text{ else } t_f \to (\text{if } t_c' \text{ then } t_t' \text{ else } t_f')_{(t_c ? \gamma_t : \gamma_f) \wedge \gamma_c}} \text{ (E} - \text{CONDSYM)}$$

Constrained terms and values:

$$\frac{t \to t'}{t|_\gamma \to t'|_\gamma} \text{ (C} - \text{TERM)}$$

$$\frac{t|_{\gamma_1 \wedge \gamma_2} \to t'}{(t|_{\gamma_1})|_{\gamma_2} \to t'} \text{ (CS} - \text{TERM)}$$

$$\frac{\text{freevars}(\gamma) = \emptyset}{v|_\gamma \to v} \text{ (C} - \text{VAL)}$$

Figure 3-1: Evaluation.

describe how to evaluate operations on symbolic expressions, and CS-TERM, which describes how to combine constraints.

The CONCRETIZE rule says that the concretization of the symbolic abstraction $\sigma y.t$ involves first evaluating t to some constrained term $t'|_\gamma$, fully evaluating the constraints, finding an assignment $y = c$ such that $y = c$ is consistent with the evaluated constraint γ', and then evaluating the term $t'|_{\gamma'}$ with the substitution $y \mapsto c$ to get the final result. This rule provides the only nondeterminism in the semantics; the nondeterminism comes from finding a satisfiable assignment for y. In this rule we assume the existence of a SAT function which determines whether a set of constraints is satisfiable. The SAT function is defined on conjunctions of arithmetic, boolean, and conditional expressions containing integers and variables. All expressions involved in the conjunction must have boolean type and all variables must have integer type.

We apply a function by evaluation the function body, evaluating the argument, and applying the function to the argument. The APP rule reduces the function being applied; if we have either a λ-expression or a conditional λ-expression we can use the APPABS or APPSYM rules to directly apply the function by first evaluating the argument and then performing substitution. Evaluation is call-by-value.

Evaluating arithmetic and boolean expressions and conditional expressions involves evaluating the arguments and pushing the constraints to the top-most level. Evaluation either joins constraints by conjunction or creates a ternary constraint operation which is of the form $t_c?\gamma_1 : \gamma_2$ and means $(t_c \wedge \gamma_1) \vee (\neg t_c \wedge \gamma_2)$.

We also have the C-TERM rule, which says that the evaluation of a constrained term involves evaluating the term body. The CS-TERM rules says that if we have a constrained term with a constraint on it, we combine the constraints using conjunction. The C-VAL rule says that if we have a constrained *value* with no free symbolic variables in the constraint, we can strip the constraint. Note that if there are free symbolic variables, we need to keep the constraint for when we concretize the expression.

From these rules we see that this evaluation of λ-LogLog adds the following to the standard β-reduction rules of the λ-calculus: propagating constraints, evaluating

the constraint before concretizing a σ-bound variable, and invoking the satisfiability oracle to find a suitable assignment to σ. The overhead from constraint propagation comes from the substitutions during program evaluation and the operations required to combine constraints for evaluating function applications, arithmetic/boolean operations, and conditions.

3.2.2 Simple typing to ensure normalizing expressions

We can enhance the semantics with a standard type system that supports recursive functions to ensure that all well-typed expressions are normalizing.

3.2.3 Constraint propagation correctness

We have the property that for a σ-bound variable y, all constraints relevant to y apply in the concretization of y. Since all σ variables become concrete values in the concretization of σ-expressions, evaluation of the term body of constrained terms proceeds analogously to evaluation in the λ-calculus. Like in evaluation of the λ-calculus, after an evaluation $t \rightarrow t'$, a λ-expression can only exist in the body of t' if t' is a λ-expression. For an evaluation $t \rightarrow t'|_\gamma$, γ captures all constraints in the body of t except those inside λ-abstractions. Because of these two properties, for an evaluation $\sigma y.t \rightarrow \sigma y.(t'|_\gamma)$ where t does not have a function type, γ captures all constraints in the body of t.

Chapter 4

LogLog language design

We have built a programming language LogLog on top of the λ-LogLog execution model, making design decisions based on ease of programming, ease of reasoning about the program, and ease of writing programs with good performance. The key decisions we made in LogLog are as follows:

- In LogLog, the programmer introduces constraints by associating them with type declarations. This provides syntactic sugar for the programmer to introduce constraints and also facilitates reasoning about symbolic expressions.
- The LogLog programmer introduces can use the **symbolic** keyword to introduce symbolic abstractions. This allows programmers to introduce symbolic variables that are scoped for the rest of the computation and concretized after the program finishes evaluating result term body.

4.1 Abstract syntax

We show the abstract syntax in Figure 4-1. At the top level, a program consists of type declarations and data declarations. The goal of evaluation is to produce a value from the data declarations. This data can have concrete and symbolic components. The programmer declares constraints to associate with symbolic values by associating them with types. During execution, the runtime system uses type and type-constraint

$$
\begin{array}{rcl}
\textit{Type}\ (\tau) & = & \textbf{int} \\
& | & \textbf{bool} \\
& | & \textbf{list}\ \tau \\
& | & \textbf{record}\ n_\tau \\
& | & \tau_1 \rightarrow \tau_2 \\
\textit{Var}\ (v) & = & n \\
& | & n\ .\ n_f \\
\textit{Value}\ (\nu) & = & p \\
& | & \textbf{record}\ \tau\ \{\ n = e\ (;\ n = e)^*\ \} \\
& | & \textit{List} \\
& | & \textbf{fun}\ v : \tau \Rightarrow e \\
\textit{Exp}\ (e) & = & \textbf{app}\ e_1\ e_2 \\
& | & e_1\ \textit{Arithop}\ e_2 \\
& | & e_1\ \textit{Boolop}\ e_2 \\
& | & \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_f \\
& | & \nu \\
& | & \textbf{symbolic} \\
\textit{ConstraintDecl}\ (\gamma_d) & = & \textbf{forall}\ n\ \textbf{in}\ n_r\ .\ \gamma_c \Rightarrow \gamma_d \\
& | & e \\
\textit{TypeDecl}\ (d_\tau) & = & \textbf{rtype}\ n_\tau\ \{\ n_f : \tau_f\ (,\ n_f : \tau_f\ )^*\ \} \\
& | & \textbf{type}\ \tau_{new} = \tau \\
& | & d_\tau\ \textbf{with}\ \gamma_d \\
\textit{DataDecl}\ (d_d) & = & n_d : \tau_d := e \\
\textit{Program} & = & d_\tau^*\ d_d^*
\end{array}
$$

Figure 4-1: Partial LogLog abstract syntax for expressions. Primitives p consist of integers and booleans. Variables can either be identifiers n or field accesses.

information to propagate constraints for symbolic expressions. These constraints are later used when making symbolic expressions concrete.

4.1.1 Concrete and symbolic data

We have the **symbolic** expression, which introduces a new symbolic variable. If we have a continuation k expressing the rest of the computation at a point we evaluate a **symbolic** expression, we evaluate the expression by introducing a new variable v and yielding $\sigma v.k\ v$. This assumes we have the computation in continuation-passing style. Concretizing of all symbolic variables occurs at the end of the computation.

4.1.2 Types and type-level constraints

Type declarations allow the programmer to declare new record types, type aliases, or associate types with *constraint functions*. The requirement for a constraint function γ_d associated with type τ is that it has type $\tau \rightarrow \textbf{bool}$. We use the resulting boolean constraint for concretizing if v is symbolic.

If we have a type τ and we declare a new type τ' with constraint γ, we have the subtyping relationship $\tau' <: \tau$. For concrete values, τ and τ' are equivalent types. Since we know symbolic expressions of type τ' satisfy the constraints declared with τ, whenever we need an expression of type τ we can also use an expression of type τ'.

4.2 Introducing and eliminating constraints

The evaluation introduces constraints by looking up type-level constraints of symbolic expressions. The evaluation eliminates constraints when evaluating the final result through concretization.

4.2.1 Looking up type-level constraints

In the LogLog framework, the runtime associates constraints with terms based on what the user has defined. When evaluating $t : \tau$, where τ is associated with some

constraint function $f_\gamma : \tau \rightarrow \text{bool}$, we get the evaluation

$$t : \tau \rightarrow t'|_{f_\gamma(t')} : \tau.$$

For instance, suppose t' were the census data record from our introductory example, where v0 is a symbolic variable:

census_data_imputed { name = 0; age = 54; married = v0; spouse = 1; ... }

Recall that we had the following constraint:

```
type census_data_imputed = census_data
  with (r) { r.spouse >= 0 implies r.married }
```

In applying the constraint to the record, we would add the constraint r.married = true to our constraint environment since the record's spouse field is a valid (positive) user ID.

When there is a data structure containing values, for instance a list of census records, the runtime will iterate over the data structure to apply constraints to each element. The programmer can also declare constraint functions on the list type, in which case the constraint function is responsible for iterating over the list.

4.2.2 Concretizing the final result

At some point in the computation, we will have fully evaluated everything except for the σ-bound variables, so we will have something of the form

$$\sigma v_0.\sigma v_1.\sigma v_2.t|_\gamma,$$

where γ encapsulates constraints on all of the σ-bound variables. At this point we can assign values to all of the symbolic variables based on γ to yield the final (concrete) result.

Chapter 5

Implementation

The main goal of this chapter is to describe how we bound memory usage in our LogLog implementation. We describe the interpreter architecture, interpreter algorithms, run-time garbage collection, and our experience optimizing the interpreter.

5.1 Bounding space

Space, rather than time, is the most important thing to bound. As we saw with the evaluation rules, running time introduces a constant factor of overhead to term evaluation time, plus the amount of time it takes to evaluate the final constraints for each symbolic abstraction. On the other hand, the sizes of the symbolic expression and final constraint are a function of the number of operations, conditionals, and function applications. For each variable substitution in a function application, this size grows by a factor of the number of times the variable appears. Without shared state, the space could grow exponentially.

We can observe, however, that since substituted expressions are immutable the number of *unique* expressions grows linearly with the number of function applications. Thus common-subexpression elimination can limit the space expansion to be linear in the number of applications. We use the structural hashing techniques performed by SMT solver such as UCLID [16] and STP [13].

5.2 High-level overview

The system includes a frontend that drives the interpreter loop, a backend runtime system that performs optimizations and garbage collection, and an SMT constraint solver for resolving linear constraints. To reduce space overhead we use eager simplification, expression sharing, and eager garbage collection in storing expressions and constraints.

We use OCaml for the frontend, C++ for the backend, and the SMT solver Yices for constraint solving. The architecture was driven by the tradeoff between ease of implementation and need to control memory usage.

5.3 Interpreter architecture

The frontend performs preprocessing, drives the evaluation loop, and performs type-level constraint applications. The backend stores evaluation state and performs eager optimizations to reduce the size of the state. The oracle resolves constraints.

5.3.1 Frontend

The frontend evaluation environment stores a mapping of names to type constraints, variable names to types, type names to type-level constraints, and function names to function definitions.

5.3.2 Backend

The backend is a runtime system that keeps track of 1) the expressions being evaluated and 2) the corresponding constraints.

The runtime system stores expressions as directed acyclic graphs of *nodes*. Atomic nodes are *constants*, *relations*, *records*, and *symbolic variables*. There are also nodes corresponding to *arithmetic expressions*, *boolean expressions*, and *ternary expressions*, which involve a guard, a true case, and a false case.

The runtime is responsible for keeping the following state:

- A stack that stores information about expression nodes. The frontend communicates with the backend through this stack.
- A variable environment mapping front-end variables to expression nodes.
- A vector of symbolic nodes to keep track of which variables need to be concretized.
- A vector of assertions as the constraint environment.

Expression nodes

Our backend uses different kinds of nodes that inherit from a parent ExpressionNode class. ExpressionNode objects store a unique ID, pointers to parents (which are NULL by default), serialization functions, and additional information relevant to the type of node it is. We have expression nodes for boolean operations (AND, OR, NOT, GT, GE, LT, LE, EQ, NEQTY), arithmetic operations (PLUS, TIMES, DIV NEG), atomic values (CONST, RELATION, RECORD), symbolic values (SYM), and ternary expressions (TERNARY).

As an optimization, we store a tagged union of expression nodes and 64-bit integers on the stack so that we don't have the overhead of creating ConstNode objects. We eagerly simplify arithmetic/boolean operations involving constants.

Symbolic nodes

The backend gives unique ID's to symbolic nodes and keeps a vector of pointers to them. Symbolic nodes can be created in the frontend or when the backend encounters missing values when reading relations from file.

5.3.3 Solver and interface

We use the Yices SMT solver [29] to resolve constraints. The backend calls the solver by writing constraints to file; the solver does not store state between calls. The

backend reads the solver output from standard input and updates the symbolic nodes to be constant nodes with their newly assigned values.

5.3.4 Frontend interface to backend

The frontend interacts with the backend by passing arguments to backend functions (to specify control flow and communicate constants), by pushing and popping nodes onto the stack (to share expression nodes), and by reading backend return values.

5.4 Algorithms

5.4.1 Frontend

The frontend drives the interpreter loop. It stores types and function declarations while the backend stores relations and the value environment.

The frontend drives the interpreter loop. The top-level evaluation loop function has the following type signature:

```
evaluate : Evalenv.evalenv ref -> Ast.topdecl list -> unit
```

Given a list of top-level declarations and a reference to a frontend evaluation environment, the function modifies the environment according to the declarations. This can include binding types, binding type-level constraints, and binding global values.

Below we show the code for evaluating top-level value assignments:

```
match decl with
| Ast.AssignDecl (name, ty, rhs) ->
    Evalenv.bindTypeEnv env name ty;
    (match rhs with
    | Ast.RhsExp exp ->
        processTopExp name exp ty;
        Evalenv.bindValueEnv env (Ast.Var name)'
        ProcessConstraints.addTypeConstraints env name
    | Ast.Concretize exp ->
        Oracle.concretize env (ref exp) ty;
```

```
        Evalenv.bindValueEnv env (Ast.Var name))
| ...
```

If the top-level declaration is an assignment to an expression, we call the evaluation loop and then apply type level constraints; otherwise, we concretize.

Expression evaluation

The evaluation loop makes nodes in the back-end and puts the result onto the backend stack. It has the following type signature:

```
evalExp : Evalenv.evalenv ref -> Ast.exp ref -> Ast.typerep -> unit
```

Given an environment, and expression, and its type, the evaluation function pushes a resulting node onto the backend stack by communicating with the backend. For instance, we show the case for evaluating arithmetic operations below:

```
| Ast.Arithop (op, e1, e2) ->
    evalExp env e1 Ast.IntType;
    evalExp env e2 Ast.IntType;
    BackendInterface.basicArithop (AstBackend.get_arithop op)
```

For an arithmetic operation with operator op and arguments e1 and e2, we first evaluate e1 and push it onto the stack, evaluate e2 and push it onto the stack, and then we call the backend interface function with the operator. The backend function, which expects the operands to be on the stack in this order, pops the operands and creates an operator node on the stack.

Constraint application

We apply constraints to values of relation or type. The function has the following type signature:

```
addTypeConstraints : Evalenv.evalenv ref -> Ast.varname -> unit
```

Given an environment and a name, we add type constraints for the variable in the backend. Note that we only add type constraints when we bind a variable at the top level.

We do various things while applying constraints to make things more efficient. When iterating through a relation, we apply constraints only when one of the implicated values is symbolic. To avoid building constraints and then throwing them away, we first determine whether a constraint could have symbolic portions before constructing it. If a constraint is predicated by a condition (for the matching constraints), we also evaluate the condition and only construct the constraint if the condition is true or symbolic.

5.4.2 Backend

Calling optimizations

Whenever we bind an object in the environment, we call the simplification and common sub-expression elimination optimizations.

The backend invokes simple optimizations when creating nodes and whole-expression optmizations when storing variables. Simple optimizations involve flattening constants at the operator level; whole-expression optimizations involve using the common-subexpression tables and more involved constant propagation.

The rate at which we perform optimizations balances the desire to optimize eagerly to minimize space usage and the overhead of optimization per expression.

Whole-expression optimizations

We have two optimization passes: one for basic constant propagation and one for common sub-expression elimination.

We perform basic constant propagation that simplifies arithmetic and boolean expressions as much as possible, looking across levels for expression trees.

The common sub-expression we store are 0 and 1 constants and to non-constant expressions corresponding to bound variables. We have a string hash table where we hash pointers to expressions using the type of the expression, its unique ID, and other relevant information.

Reference counting garbage collection

We perform reference counting garbage collection in order to do it as eagerly as possible so that we do not run out of space. We produce garbage mostly as a result of performing optimizations.

We will define what it means for a variable to be live and explain an algorithm that satisfies the following properties:

- An expression is *live* if the front-end can still access it.
- An expression is dead if and only if its reference count is zero. If an expression is live, its reference count is positive.
- If an expression is dead, our backend will reclaim the memory.

An expression is *live* if there is still a pointer to the expression. An expression can be live in the following ways:

- There is a pointer to the expression on the stack.
- The expression is bound in the variable environment.
- The expression is an ancestor of some other expression.
- There is a pointer to the expression in our common sub-expression tables.

Whenever a node is set as the parent of another node, pushed onto the stack, or bound as a variable, we increment the reference count.

A node can only become garbage when we perform optimizations. Thus during optimizations, whenever we replace a node we are careful to

5.5 Experience notes

We learned quite a bit from trying to implement a prototype interpreter that was sufficiently efficient with respect to time and space.

We initially implemented the interpreter in OCaml with a Yices backend using standard functional idioms. We first optimized that interpreter and then re-implemented the back-end using C++. We switched for two reasons: it was becoming difficult to get performance gains from the OCaml without reasoning about the OCaml compiler and we had some existing C++ infrastructure supporting efficient processing and storage of symbolic expressions.

5.5.1 Performance wins

The biggest performance win, both in the OCaml implementation and in the mixed implementation, came from getting almost constant-time access to all records sharing a field value. For instance, if we are applying constraints that depend on matching on field f, instead of looking through the relation for matching field f it is much more efficient to store the records as a hash table of lists keyed by field f.

OCaml implementation

We saw performance gains from using references instead of values as much as possible, getting rid of closures, and compiling to native code rather than byte code. We also saw performance gains from eager optimization of symbolic expressions.

Mixed implementation

We needed to implement eager reference-counting garbage collection in the C++ backend in order to not run out of memory on processing large relations. A significant speedup came from noticing that rather than storing all nodes, we can keep a stack of nodes and only store bound nodes in a map. We also saw good speedups from using a tagged union of expression node pointers and integers, since most of the values we saw were actually integers. Because the tagged union structure was small enough to store directly rather than allocating on the heap, we were able to avoid the allocation and deletion overhead.

5.5.2 Ease of implementation vs. runtime performance

Our OCaml interpreter was initially easy to implement and maintain. The initial implementation, which occured in tandem with making language design decisions, took one person-month. As the OCaml implementation became more optimized, however, maintenance became more time-consuming–the complexity came from having to reason about the order of stateful operations such as environment updates.

The C++ backend, which exhibits the same behavior as the initial OCaml interpreter, was fairly painful to implement. It took almost 3 person-months; much of this difficulty came from the fact that we were modifying a large existing platform of sparsely documented research code.

Chapter 6

Case Study: Processing Real Census Data

We conducted a case study implementing documented imputation strategies on hundreds of thousands of records from the U.S. Census Bureau. Our benchmarks are based on the documented imputation strategies. We also compare LogLog to SQL and Python in terms of both programming and efficiency of the resulting program.

We show the following:

- Data processing is a suitable domain for specification-enhanced execution. While there are many real-world data items that may be missing, only a small fraction of them are missing in a given data set.
- LogLog provides a good programming interface for expressing the documented imputation strategies.
- Constraint solving takes up a negligible portion of the total running time.
- Constraint derivation is reasonably fast.
- Symbolic execution does not introduce more runtime overhead than standard execution.

This suggests that this approach is not only useful but feasible for principled data cleaning of real world data.

Month/year	Household		Demographic	
	Range	Mean	Range	Mean
Jan. 1997	0.16 - 5.84	0.41	0.08-9.2	1.35
Jan. 2004	0.01-3.42	0.56	0.12-19.0	2.41

Table 6.1: Imputation rates for different CPS fields by percent [26]. *Household edits* involve hot deck imputation between households. *Demographic edits* involve employing relational, longitudinal, and hot deck imputation based on demographic variables such as marital status, parents, and spouse.

6.1 Current Population Survey and imputation

We use data from the U.S. Census Bureau's Current Population Survey (CPS), a monthly survey sampling about 50,000 households used by government policymakers, students, and academics [24]. Imputation strategies have been shown to be important in affecting research conclusions in this domain [3]. Specifically, we use the Annual Social and Economic (ASEC) March supplement, which provides *microdata*, or survey data where the unit of observation is individuals, families, and households [25]. This data contains relationship information about people within families and households and social information such as the number of children in the household enrolled in public school. The 2009 supplement contains 392,550 records.

The Census Bureau documents standard procedures for reconstructing answers from nonresponse from other available information [25]: *relational imputation* involves inferring the missing value from other characteristics on the person's record or within the household; *longitudinal edits* involve looking at the previous month's data; *"Hot deck"* imputation infers the missing value from a different record with similar characteristics. While there are many ways in which imputations may need to occur, the number of imputations is low relative to the size of the data set, as we show in Table 6.1.

The supplement data sets contain preprocessed data that is annotated with a set of fields containing *imputation flags*, which indicate how certain fields have been imputed. These flags may either be binary imputed/not imputed, or they may contain

Household	Line	Age	Mar.	Spouse	MI	SI
26974	11	00	9	00	0	0
26974	13	06	0	70	0	0
26974	05	27	**1**	**07**	4	2
26974	07	32	1	05	0	0

Table 6.2: View of the March 2009 CPS data showing the following columns for a subset of entries from household 26974: household ID, line number, age, marital status, spouse line number, marital status imputation flag (MI), and spouse imputation flag (SI).

more information, such as "Allocated from hot deck" or "Allocated from spouse" [24]. In Table 6.2 we show some entries from a household where the marital status information for one of the household members has been imputed. We can see that the marital status and spouse line number for the person on line 5 came from the person on line 7.

6.1.1 Experimental setup

We use the following columns of the CPS data:

- *Household ID.* The March supplement assigns a unique identifier to each household.
- *Individual line number.* The data identifies individuals according to their line number with respect to a household.
- *Age.* From the imputation flags and surrounding data, it appears to be imputed from reference tables from previous years.
- *Marital status and spouse line number.* The documentation says that these values are imputed from relational information. The spouse line number is with respect to the same household ID. Marital status can be integer values $0 - 9$ indicating statuses including "single, never married" and "widowed."

```
(* CPS data record type. *)
rtype cps_data
  { household    : int
  ; line_no      : int
  ; age          : int
  ; marital      : int
  ; spouse_no    : int
  ; hot_lunch_no : int}
type cps_rel = cps_data relation

(* RELATIONAL IMPUTATION *)
type m_data_inferred_rel = cps_rel
  with (rel) {
    forall r, r' in rel :
      ((r.household == r'.household) and
       (e.spouse == r'.line_no)) implies ((r.line_no == r'.spouse) and (e.married == r'.married))
  }

(* Load data and take conditional sum. *)
in_rel : m_data_inferred_rel = load data.txt
result : int = concretize (sum_married in_rel)

(* LONGITUDINAL IMPUTATION *)
(* Load reference relation. *)
reference_rel : cps_rel = load 'ref.txt'

(* Define data type for longitudinal imputation. *)
rtype age_data_inferred_rel = cps_rel
  with (rel) {
    forall r in rel : exists r' in reference_rel forsym r.age :
      ((r.household == r'.household) and
       (r.line_no == r'.line_no)) implies (r.age == r'.age)
  }

(* Load data and sum over age field. *)
in_rel : age_data_inferred_rel = load 'data.txt'
result : int = concretize (cun_ages in_rel)

(* HOT DECK IMPUTATION *)
type lunch_data_inferred = lunch_data relation
  with (rel) {
    exists r, r' in rel forsym r.hot_lunch_no:
      (r.household == r'.household) implies (r.hot_lunch_no == r'.hot_lunch_no)
  }
```

Figure 6-1: Relevant code for imputation tests.

- *Number of children in household who get hot lunch.* Fields like this seem to be imputed from surrounding values that are similar: in this case, this value can be imputed from another individual from the same household.

We tested the performance of the LogLog implementation on the following kinds of imputations:

1. *Relational imputation.* We impute missing marital status and spouse fields for a record with household ID h line number ℓ by looking at whether there is another record with household h listing ℓ as the spouse line number.

2. *Longitudinal imputation.* We impute missing ages by looking up the value in a reference table.

3. *"Hot deck" imputation.* We impute a missing hot lunch number for a record with household ID h by using a value from another record with household ID h.

4. *Combined imputation.* We combine these three imputation strategies, creating a type with all three constraints and performing a sum over the age field.

We show the relevant code from these tests in Figure 6-1. Note that since longitudinal and hot deck imputation have constraints necessarily apply to concrete values, we use **exists** rather than **forall**.

For each example, we created a data set with missing values by examining the relevant imputation flags. For example, for the relational imputation test we use the imputation flags for the marital status and spouse line number to set the implicated items as missing while leaving the other items intact. For the longitudinal imputation example, we used the original data set as the reference table. The input data for the "combined imputation" example has missing items in all relevant columns.

6.1.2 Results

We ran the tests on increasingly large subsets of the data with the following total percentages of records with missing items:

Total records	# bad	Relational			Longitudinal		
		Cnsts.	Yices	Total	Cnsts.	Yices	Total
8,192	66	0.02	0.000	0.08	0.04	0.000	0.08
16,384	152	0.06	0.000	0.15	0.07	0.000	0.16
32,768	344	0.25	0.000	1.28	0.14	0.000	0.33
65,546	1007	0.28	0.004	0.64	0.30	0.000	0.69
131,072	1990	0.57	0.000	1.30	0.64	0.000	1.46
262,144	3745	1.16	0.004	2.62	1.32	0.000	3.02
392,550	5693	1.81	0.004	4.09	2.04	0.000	4.62

Total records	# bad	"Hot deck"			Combined		
		Cnsts.	Yices	Total	Cnsts.	Yices	Total
8,192	66	0.02	0.000	0.06	**0.10**	**0.000**	0.16
16,384	152	0.06	0.000	0.12	**0.20**	**0.000**	0.32
32,768	344	0.12	0.000	0.26	**0.42**	**0.000**	0.67
65,546	1007	0.25	0.000	0.54	**0.88**	**0.000**	1.40
131,072	1990	0.53	0.000	1.11	**1.82**	**0.000**	2.87
262,144	3745	1.09	0.000	2.30	**3.86**	**0.000**	6.04
392,550	5693	1.67	0.000	3.54	**5.82**	**0.000**	9.04

Table 6.3: Times (in seconds) for 1) deriving constraints on the input with incomplete values (Cnsts.), 2) solving the Yices constraints (Yices), and 3) running the interpreter on the entire program (Total). The time for deriving and solving constraints makes up a small fraction of the total running time. We show the total number of records on the left, along with the total number of missing records in the data set combining all missing items.

Total records	# bad	Combined no missing	Combined		
			Cnsts.	Eval.	Total
8,192	66	0.06	**0.10**	0.06	0.16
16,384	152	0.12	**0.20**	0.12	0.32
32,768	344	0.24	**0.42**	0.23	0.67
65,546	1007	0.48	**0.88**	0.48	1.40
131,072	1990	0.98	**1.82**	1.05	2.87
262,144	3745	1.99	**3.86**	2.18	6.04
392,550	5693	3.01	**5.82**	3.22	9.04

Table 6.4: While constraint evaluation introduces an expected factor of two overhead, symbolic evaluation does not introduce signifcant overheads.

Relational	Longitudinal	"Hot deck"	Combined
0.433%	1.445%	0.588%	1.450%

We ran the programs on an machine with an Intel Core 2 Quad Q9650 processor (3.0 GHz, 12M L2 cache, 1333MHz FSB) running 64-bit Linux. We show the running times of the LogLog interpreter on our data sets in Figure 6.3. The results show that 1) the time it takes to apply the constraints grows roughly linearly and 2) the time it takes to solve the constraints is low. We expect the constraint derivation and solving to scale: Yices is good at solving unquantified constraints, and we derive constraints linear in the number of missing records.

According to these results, expression evaluation is much slower than constraint application or constraint solving. This is because we have not done much to optimize the symbolic evaluation; the interpreter performance does not reflect limitations on the speed of the symbolic evaluation. In the longitudinal and combined examples the interpreter slows down quite a bit. This is because we load the reference relation into memory and thus incur more garbage collection overhead during evaluation.

6.2 Comparison with other languages

We found that LogLog provides usability advantages over both SQL and Python and that LogLog is much faster than Python on the census data benchmarks.

6.2.1 SQL

As expected, we were able to perform the imputation much faster in SQL. We found, however, that while SQL seems to have similar language features for specifying each individual imputation strategy, getting the imputation right involves carefully managing the order of updates to make sure the necessary values propagate where they need to go.

The target SQL query we want to write is straightforward:

```
(select sum(age) from combined_data where married = 1 and hotlunch = 0);
```

In order to do this, however, we need to perform imputation on the missing fields.

Doing the imputation in SQL involves creating the appropriate tables to fill in the missing values. We can run something like the following query runs in 0.10 seconds:

```
update combined_data, combined_data_ref
  set combined_data.age = combined_data_ref.age + 1
  where combined_data.age is null
    and combined_data.household = combined_data_ref.household
    and combined_data.line = combined_data_ref.line;
```

Imputing the "married" field in an analogous takes 0.03 seconds; imputing the "hot lunch" field takes 0.05 seconds.

While SQL provides good language support for specifying individual imputation strategies, it is up to the programmer to manage the order of the updates and other dependencies. For instance, consider the case when we are imputing age from the previous year, but the previous year's age is also missing. While LogLog allows us to write a set of constraints that we can reuse and have the runtime system apply only when we need them, in SQL we would need to manually consider all cases and manually execute the imputations by hand, in the right order. The LogLog strategy

allows more flexibility in specifying application-specific imputation strategy, as it does not require the programmer to maintain multiple versions of the data and manually track versioning and dependencies.

6.2.2 Python

We implemented the benchmark examples in Python, which is similar to LogLog because it is another interpreted language not specifically optimized for data processing. We found LogLog to be both more usable and more efficient than Python for the benchmark examples.

As expected, we found it easier to write the benchmarks in LogLog than in Python. We first wrote a version of the imputation where we imputed all missing values in a record before processing the aggregation function and then we made this function more optimal by short-circuiting some evaluations. We show the fastest version of the program in Figure 6-2; including comments it is almost 100 lines of Python.

The running times in Figure 6.5 show that while Python performs better when the constraints are simple, LogLog outperforms Python when we need to impute multiple values. While the LogLog program runs over all examples in 9.04 seconds, the unoptimized Python program takes 256.18 seconds and the faster version takes 86.03 seconds.

Total	Relational		Longitudinal		"Hot deck"	
records	LogLog	Python	LogLog	Python	LogLog	Python
8,192	0.08	0.02	0.08	0.02	0.06	0.02
16,384	0.15	0.03	0.16	0.03	0.12	0.03
32,768	1.28	0.06	0.33	0.06	0.26	0.07
65,546	0.64	0.13	0.69	0.14	0.54	0.13
131,072	1.30	0,33	1.46	0.31	1.11	0.30
262,144	2.62	0.72	3.02	0.72	2.30	0.68
392,550	4.09	1.29	4.62	1.28	3.54	1.17

Total	Combined		
records	LogLog	Python/init.	Python/opt.
8,192	0.16	0.08	0.02
16,384	0.32	0.30	0.13
32,768	0.67	1.12	0.42
65,546	1.40	6.78	2.13
131,072	2.87	32.30	9.98
262,144	6.04	115.63	38.42
392,550	9.04	256.18	86.03

Table 6.5: Python running times for comparable examples.

```
def sumInputRelCondAge():
  age_sum = 0
  for entry in input_rel:
    married = False

    # Figure out if the current person is married.
    if entry[3] == "1":
      married = True
    elif entry[3] == "!!":
      # If the entry is missing, loop over the list of records in the same
      # household to find if someone lists the current person as their spouse.
      for housemate in input_rel_by_household[entry[0]]:
        married = housemate[4] == entry[1]

    # Figure out if the current person has anybody in the household getting hot
    # lunch.
    gets_hot_lunch = False
    if entry[5] == "!!":
      for housemate in input_rel_by_household[entry[0]]:
        housemate_hotlunch = housemate[5]
        if housemate_hotlunch != "!!":
          gets_hot_lunch = int(housemate_hotlunch) > 0
    else:
      gets_hot_lunch = int(entry[5]) > 0

    if married and gets_hot_lunch:
      # Now the age might be missing too.
      if entry[2] == "!!":
        age = 0
        for housemate in ref_rel_by_household:
          if housemate[2] != "!!":
            age = int(housemate[2])
      else:
        age = int(entry[2])
      age_sum = age_sum + age
  print age_sum
```

Figure 6-2: Imputation code in Python.

Chapter 7

Related work

In this chapter we discuss related work in aspect-oriented programming, executing specifications, and program analysis involving. We also describe how the functionality LogLog provides compares with the state of the art in data processing.

7.1 Aspect-oriented programming

This work is related to aspect-oriented programming [14], which proposes a programming technique that helps isolate program aspects for specific functionality such as error and failure handling. Aspect-oriented programming involves program aspects that executes when control reaches a certain point. There are two issues with aspect-oriented advice: 1) it may interfere with normal program function and 2) it relies on traditional compiler technologies and thus rather than permeating the computation, the advice takes responsibility for subcomputations. Dantas and Walker address the first issue by showing how to apply aspect-oriented programming to use a model of "harmless advice" where they propose a framework where aspects obey a weak non-interference property and present an information flow security case study [6].

The specification-enhanced execution model solves the interference problem by having the constraints only apply to a well-specified subset of expressions during execution. Our model resolves the second issue by proposing a novel mixed execution model rather than relying on traditional compiler technologies.

7.2 Programming with specifications

7.2.1 Executing specifications

Carroll Morgan extends Djikstra's guarded command language for predicate transformer semantics with specification statements, which specify parts of the program that are "yet to be developed" [18]. Morgan's goal is different from ours in that while the specification statement executes as a subprocedure, our approach allows the specification to govern part of the execution alongside the rest of the program.

7.2.2 Constraint-based programming

Our goals are similar to those of Kaleidoscope'90, a "constraint imperative programming language" that combines imperative and constraint-based programming paradigms into a single language [4]. Kaleidoscope aims to provide full support for constraint-based programming within an imperative model, making it difficult to provide performance guarantees. Also, Kaleidoscope follows a reactive model, allowing constraints to introduce control dependencies between unrelated pieces of code.

7.3 Program analysis and repair

Our domain-specific use of goal-oriented programming for handling exceptional cases is similar to the approach of Demsky's work on data structure repair [7]. While this work uses goal-oriented programming for on-the-spot repairs, our programming model uses constraints for determining program execution over a longer period of time.

This has the flavor of the "let the program run" philosophy of acceptibility-oriented computing [22], but while acceptibility-oriented computing is motivated by the lack of a specification, our model assumes the desire to adhere to a specification.

We employ techniques familiar to the program analysis community to yield consistent results: we use constraint propagation techniques similar to those used in program analysis [15, 1, 2] and our combined symbolic and concrete execution is similar to the combined symbolic and concrete execution in concolic testing [23].

7.4 Data processing

7.4.1 Parsing

Like PADS system for processing ad hoc data [9, 5, 27, 17], we are addressing the difficulties in handling real-world data. While PADS addresses the difficulties of handling syntactic corner cases (parsing), we are concerned with semantic corner cases (processing well-formatted data that is not semantically well-formed).

7.4.2 Constraint databases

LogLog differs from constraint databases, which 1) use constraints for resolving queries and 2) do not use constraints to modify the value of data [8, 21].

7.4.3 Data cleaning

The state-of-the-art for data cleaning is based on the Extraction Transformation Loading (ETL) paradigm, in which a data cleaning tool is used to transform the data and create a cleaned data set on which further analysis can be performed. The LogLog strategy allows more flexibility in specifying application-specific imputation strategies and has the advantage that the imputation strategy is documented alongside with the program. This eliminates the need for maintaining multiple variations of the data and keeping track of which versions should be used with which applications.

Other data cleaning tools incorporate constraints to facilitate the expression of imputation strategies. For instance, Potter's Wheel provides constraints that help identify the missing data [20, 19]. Once the tool has identified missing data, the programmer can interactively specify how to handle these exceptional cases with the goal of producing a transformed data set. The Ajax framework [11], by Galhardas et. al. is based on their declarative data cleaning method [12] and provides a logic for users to specify data cleaning transformations of mapping, matching, clustering, and merging, and interacts with the user to handle the exceptional cases. All of these tools are based on the ETL paradigm and are thus different from LogLog.

Chapter 8

Conclusions and Future Work

We present *specification-enhanced execution*, a paradigm for transferring responsibility for certain aspects of the program to the runtime system. The specification-enhanced paradigm is particularly useful in aiding the programmer there are so many possible dynamic cases that the static search space is too large to perform synthesis.

In this thesis we describe the programming and execution model and LogLog, a novel programming language that support specification-enhanced execution. We demonstrate the feasibility of our programming and execution models with a case study in processing census data. We show that our implementation can process real applications on real data in seconds with negligible constraint-solving overhead. We also show that it is both less efficient and less natural to write the same programs in Python. [*Mention SQL.*]

Our results support the feasibility of 1) using LogLog for data processing and of 2) applying the specification-enhanced execution model to other domains. These results are promising for leaving programming details to runtime assistants in the future.

8.1 Future work

An immediate direction for future work is exploring applications to information flow. One aspect that programs often delegate to runtimes is information flow and access control: making sure sensitive data leaks through certain boundaries only when the

receiving end has the appropriate permissions. Access control requires functionality that may be orthogonal to program functionality, but it is very much tied to the entire program execution. It is also natural to express access control specifications in the form of simple, high-level logical constraints. Specification-enhanced execution can allow the programmer to delegate access control management to the runtime system 1) without requiring the programmer to provide heavy annotations and 2) without requiring a new compile-time or run-time framework specifically for this purpose.

We would also like to develop a framework generalizing specification-enhanced execution to control flow specifications in a way that still yields efficient programs. The vision is to develop a method for easily creating robust systems by delegating tedious and error-prone aspects of system behavior (for example, exception handling) to the runtime system.

Bibliography

[1] A. Aiken. Scalable program analysis using boolean satisfiability. *Formal Methods and Models for Co-Design, ACM/IEEE International Conference on*, 0:89–90, 2006.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2007. ACM.

[3] Richard V. Burkhauser, Shuaizhang Feng, Stephen P. Jenkins, and Jeff Larrimore. Estimating trends in us income inequality using the current population survey: The importance of controlling for censoring. Working Paper 14247, National Bureau of Economic Research, August 2008.

[4] James R. Cordy and Mario Barbacci, editors. *ICCL'92, Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, USA, 20-23 Apr 1992*. IEEE, 1992.

[5] Mark Daly, Yitzhak Mandelbaum, David Walker, Mary Fernández, Kathleen Fisher, Robert Gruber, and Xuan Zheng. Pads: an end-to-end system for processing ad hoc data. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 727–729, New York, NY, USA, 2006. ACM.

[6] Daniel S. Dantas and David Walker. Harmless advice. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 383–396. ACM, 2006.

[7] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 176–185, New York, NY, USA, 2005. ACM.

[8] Jan Van den Bussche. Constraint databases, queries, and query languages. In *Constraint Databases*, pages 20–54, 2000.

[9] Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. *SIGPLAN Not.*, 40(6):295–304, 2005.

[10] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 561–578, London, UK, 2001. Springer-Verlag.

[11] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: An extensible data cleaning tool. In *SIGMOD Conference*, page 590, 2000.

[12] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 371–380, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[13] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[15] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[16] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In *CAV*, pages 475–478, 2004.

[17] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. Pads/ml: a functional data description language. *SIGPLAN Not.*, 42(1):77–83, 2007.

[18] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.

[19] Vijayshankar Raman, Andy Chou, and Joseph M. Hellerstein. Scalable spreadsheets for interactive data analysis. In *1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 1999.

[20] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 381–390, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[21] Peter Z. Revesz. Datalog and constraints. In *Constraint Databases*, pages 155–170, 2000.

[22] Martin Rinard. Acceptability-oriented computing. In *In 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA âĂŹ03 Companion) Onwards! Session*, pages 221–239. ACM Press, 2003.

[23] Koushik Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM.

[24] The Bureau of Labor Statistics and the Census Bureau. Current population survey. `http://www.census.gov/cps/`, 2009.

[25] U.S. Census Bureau. *Current Population Survey, 2009 Annual Social and Economic (ASEc) Supplement*, 2009.

[26] U.S.Census Bureau. *Technical Paper 66, Design and Methodology: Current Population Survey*, 2006.

[27] Qian Xi, Kathleen Fisher, David Walker, and Kenny Qili Zhu. Ad hoc data and the token ambiguity problem. In Andy Gill and Terrance Swift, editors, *PADL*, volume 5418 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2009.

[28] Paul Yao. Selecting a windows mobile api - .net compact framework and win32. `http://msdn.microsoft.com/en-us/library/dd630621.aspx`, April 2010.

[29] Yices: An smt solver. `http://yices.csl.sri.com/`.