Programmer's Guide to
The Recognition Strategy Language (RSL)

Version 2.0 (DRAFT)

Richard Zanibbi[1]

March, 2011

Note on the History of RSL

The first version of RSL was designed and developed by Richard Zanibbi, Dorothea Blostein and James R. Cordy at Queen's University, Kingston, Canada in 2004. This document summarizes the re-design and re-implementation of RSL produced at the Rochester Institute of Technology by Matthew Fluet, Ben Holm and Richard Zanibbi during 2010-2011.

Contents

Chapter 1

The Recognition Strategy Language

The Recognition Strategy Language (RSL) is a domain-specific language for pattern recognition. It is implemented through translation and compilation in Standard ML (SML) using the MLton whole-program optimizing compiler. This document summarizes the second version of RSL, which is a complete re-design of the original language. This document summarizes the key features and use of the language from the programmer's perspective.

Pattern recognition applications include computer vision, speech recognition, time series analysis, and many others. RSL was designed from the viewpoint that pattern recognition algorithms may be most easily understood in terms of the decisions they make, and that they should therefore be defined in terms of decision sequences. Generally speaking, given some data such as an image, these decisions are generally answers to one or more of the following questions:

1. Where are the objects of *this* type? (Segmentation)
2. What are the type(s) of *these* objects? (Classification)
3. How are *these* objects of *these* types structured? (Parsing (relating objects))

For non-trivial applications, these questions are often easy to pose but difficult to answer. For example, it is easy to ask, 'Where are the faces in this image?', but face recognition is not a solved problem in the general case. However, if we refine our question just a little bit, instead asking: 'Where are the *bounding boxes* that contain faces in this image?', the set of possible answers is well-defined and finite (though large). RSL is designed to make it easy to express recognition decisions, and to change the functions that make decisions. We refer to this as *decision-based specification* of recognition algorithms.

In the list of recognition questions above, we see that one must define a set of locations in the data before they can be classified, that defining a set of (meaningful) locations requires a decision about the class of objects sought after, and that relating (parsing) locations requires a set of locations with assumed types. In addition, current information about structure (parsing) can be used to constrain the location and types of objects. Machine-learning techniques may also be used to jointly optimize a group of decisions. Using our face detection example, we might repeat a process of defining a set of candidate face rectangle locations, and repeatedly change these to maximize the probability that each of the final set of rectangles contains a single face, and that the region outside these boxes does not contain a face. To do this effectively, one needs to address the following additional question:

4. Which is the best value for *this/these* decision model parameter(s)? (Machine Learning)

The often complex functions that make decisions are defined separately from the main RSL program. These decision functions may be defined using different programming languages. For example, decisions

based on logical inference might be expressed in Prolog, and decisions requiring image processing might be expressed using Matlab, or C using the OpenCV library. These 'external' decision functions are functions in the host language (Standard ML), which then interface with the other languages using the ML Foreign Function Interface (FFI) or system calls. The current RSL implementation makes use of the FFI provided by the MLton compiler. RSL makes it easy to express mapping a decision or output function over the current set of interpretations, making it easier to create evaluation data, visualizations, or other useful information. The re-implementation of RSL in ML allows decision functions to be implemented in a concise functional style, with support for higher-order functions and automatic type inference (provided by SML, which is strictly typed).

In addition to decision-based specification, another central feature of RSL is the automatic creation of a history of the decisions made by a program, along with the ability to query this history. This makes it possible to determine exactly which decisions select correct and incorrect alternatives (in our example, bounding boxes for faces), and at what point(s) in the program's execution. It also makes it possible to evaluate the set of generated hypotheses within an RSL program, such as using the recall and precision of this set (known as the *historical* recall and precision of generated hypotheses).

The revised RSL also permits annotations for decision to be recored in the history, and then observed after the decision-making portion of an RSL program is complete. This is an important change, as it allows the output of arbitrary computations to be stored separately from the interpretation data, without being available to the decision process itself.

The remainder of this document briefly summarizes the use and key features of RSL.

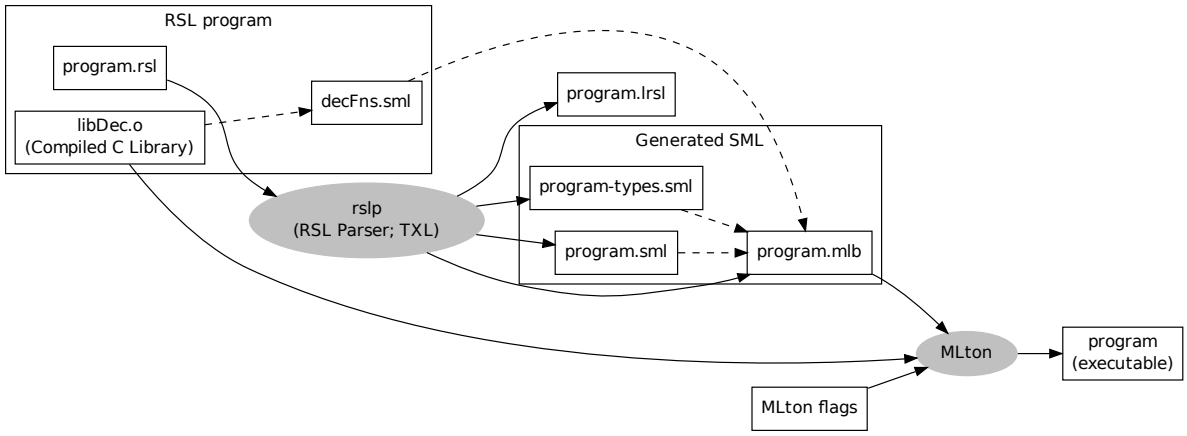
1.1 Compilation and Execution

Figure 1.1 illustrates how RSL programs are compiled and executed using the `rslc` program. `rslc` is itself a script which calls the RSL Parser (`rslp`, a TXL program), followed by the MLton Standard ML Compiler. MLton is a whole-program optimizing compiler that produces efficient executables. Invoking `rslc` without arguments produces the following:

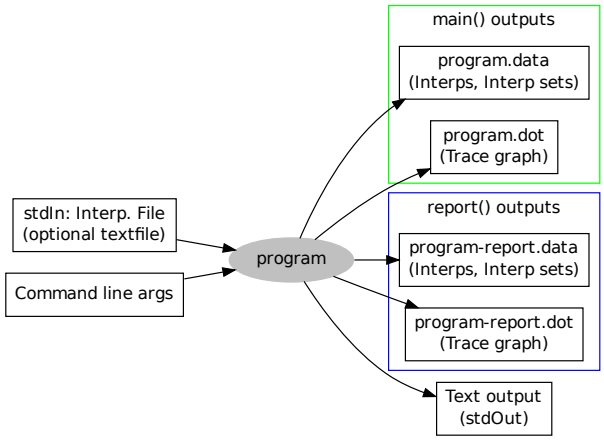
```
RSL Parser -> MLton Compiler script
Copyright (c) 2010,2011

Usage: rslc [-v] <program.rsl> [mlton-compilation-flags] [.o/.c libs]

-v: generate verbose output (.dot graph file, .data output file)

NOTE: compiler flags and .o/.c libraries may be intermixed.
      It is assumed that compiler flags do not have .o/.c
      files as values.
```

At compile time, the user provides an .rsl program (`program.rsl` in Figure 1.1) which may include one or more Standard ML files containing decision functions (`decFunctions.sml` in Figure 1.1). Any compiled C library files invoked by decision functions to the RSL compiler must be passed to `rslc`, which simply passes them on to the MLton compiler.

Given an .rsl program, the RSL parser (`rslp`, implemented in the TXL language) generates files for interpretation types and operations (shown as `program-types.sml`), the SML translation of the RSL program (`program.sml`), and a MLton build file (`program.mlb`). An additional file with the extension `.lsrl` is produced for reference (this is a modified version of the user's program), but not used during compilation with MLton.



(a) **RSL Program Compilation.** Inputs and outputs are shown using solid liners; file dependencies are shown using dashed lines. `program.lrsl` is an annotated RSL file produced before generating .sml code, and is produced for reference only. Objects files for compiled C programs may be passed to MLton (their use requires the MLton Foreign Function Interface). **The rslc program invokes both rslp and MLton.**



(b) **RSL Program Execution.** An *interpretation file* (textfile in a format defined by the programmer) may be passed on the standard input, and if provided will define the initial set of interpretations for the program (otherwise an empty set is used). **If rslc is invoked using the -v option,** then in addition to text sent to the standard output (e.g. from `print` statements), snapshots of the execution traces at the end of the `main()` and `report()` functions are produced.

Figure 1.1: RSL Program Compilation and Execution

For the main RSL program, The main program translation maps RSL functions to ML functions, and RSL operations to operations in Abstract Syntax Tree (AST) nodes, and performs some context-sensitive syntax checking for the RSL program (e.g. insuring that 'reporting' operations are not used in the main program). The output executable is produced by compiling the generated SML files with the RSL Core

library, which includes code for interpreting the generated ASTs.[1]

An executable RSL program may be used with different input files, called *interpretation files* or *i*files for short, that contain zero or more interpretations of the data to be recognized in the RSL program. These files are simply a list of interpretations, where individual interpretations are in a format specified by the programmer. A detailed description of interpretation files are provided below in Section 1.2. Running an interpretation file through an RSL program produces data on the standard output. **If rslc is run with the -v (verbose) option**, then the execution will also produce files summarizing the execution trace after the `main()` function has completed, and after the `report()` function has completed. These output files are produced only if the `main()` or `report()` functions are defined.

A Note on Sharing ifiles Between RSL Programs: An RSL program may use an ifile produced by another RSL program, provided that the interpretation type definitions are the same. This allows RSL programs to be built up from smaller programs, or for RSL programs to be broken up into a set of smaller programs that can be chained together.

1.2 Interpretation Input and Output: *ifiles* and Related Functions

An interpretation file, or *ifile* is a text file representing a set of interpretations in a list. The format is very simple, containing:

- The number of interpretations (**first line**)
- For each interpretation:

  [Interp *(number)*]
  (((Interp. data in programmer-defined format)))
  blank line

Here is a simple interpretation file, containing three interpretations, with interpretations represented by a single integer:

```
3
[Interp 1]
9

[Interp 2]
11

[Interp 3]
13
```

Note that RSL programs can automatically generate the number of interpretations, interpretation headings, and invoke user functions to write an interpretation file such as the one shown above. Users may also manually create or modify ifiles on their own (e.g. to quickly generate test cases, or try different inputs for a given RSL program).

Figure 1.2 lists the functions used in RSL to read and write ifiles (each has a default definition in `rsl/trunk/rsl_parser/AstTraceExec.sml` that has no effect). The `streamToInterp` function is used to read interpretations below the "[Interp *number*]" headings from ifiles passed on the Standard

[1] dependencies on the Core RSL library (implemented in SML) are not shown in Figure 1.1. The set of RSL Core library files may be found by looking at the .mlb file generated by the RSL parser (`rslp`).

FUNCTION	PURPOSE
`streamToInterp`: TextIO.stream → Types.Interp.t	Read single interpretation from a text stream (Standard Input or text file)
`interpToString`: Types.Interp.r → string	Create string for single (raw: Types.Interp.r) interpretation.
`hcInterpToString`: Types.Interp.t → string	Create string for single (hash-consed: Types.Interp.t) interpretation.
`annotToString`: Types.Annot.t → string	Create a string representing an annotation.

Figure 1.2: Pre-defined Interpretation Input and Output Functions. Programmers may over-ride the default implementations of these functions (e.g. in an included file `decFns.sml`). `annotToString` is used when producing output for annotations in the execution trace, but not for reading/writing interpretations.

Input, or added to the current interpretation set from a file using the `add` operation. The output function `interpToString` is used to implement the `print ifile` and `write ifile <filename>` commands, which print an interpretation file for the current set of interpretations in an RSL program to the standard output or an external file, respectively.

1.3 Example Programs

1.3.1 Hello World

As a first example, below is 'Hello World' in RSL. Note that an interpretation type must be defined (here with a single integer field), but is not used. Note that comments begin with '(*' and ending with '*)'.

```
(* Interpretation type declaration *)
interp:
    field: int

(* This is the main function. *)
fn main() {
    print "Hello World."
}
```

1.3.2 A More Complex Example

Figure 1.3 shows a more complex RSL program, which makes use of decision points. For simplicity, interpretations consist of just an integer, and an integer set. The file containing decision and other external functions (`firstExampleExternalFns.sml`) is shown in Figures 1.4 and 1.5. The first part of the external function file re-defines a number of the built-in output functions described above in ML. The second part of the file defines decision functions used to modify the current set of interpretations.

The execution trace graph corresponding to the `firstExample.dot` file produced on running the program is shown in Figure 1.6.

In the trace graph shown in Figure 1.6, nodes and edges are numbered in sequence (in hexadecimal), with nodes corresponding to decision points with an associated set of interpretations (shown in boxes with dashed arrows coming out of nodes). Edges are annotated with the operation performed, and the associated decision point label. Note that `rslp` attaches a prefix `D#_` to each label associated with a decision (given in square brackets in the .rsl program), insuring that each decision point has a unique label.

The `Init{INIT}` operation coming out of the node `N_0000` corresponds to reading interpretations from the standard input; here no data was provided on the standard input. The conditional statement `if` is broken up into nodes for the sets of interpretations for which the conditional test is true and false, and then the interpretations produced in the true/false branches are merged in node `N_0011`. Note also that some nodes share identical interpretation sets in the graph, for example the initial empty set of interpretations remains empty when the program tries to read additional interpretations from the standard input (node `IS_0000`, the empty set of interpretations).

The file `firstExample.data` illustrates the type of output produced in .data files. Note that this provides additional information related to the interpretations and interpretation sets generated by the program.

```
(* Include (SML) decision functions from another file *)
inc "firstExampleExternalFns.sml"

(* Interpretation type. *)
interp:
  IntField: int
  IntSet: int set

(* Main function. *)
fn main() {
  (* Add initial interpretationt using function initInterp. *)
  [ I ] munge: initInterp

  (* Update interpretations *)
  [ A ] update: decA
  [ B ] update: decB

  (* Accept interpretations for which decC returns true *)
  [ C ] if decC {
    accept
  } else {
    reject
  }
}
```

Figure 1.3: Example RSL file

```
(* OUTPUT FUNCTIONS *)
val annotToString = fn (a : Annot.t) => case a of
                                          NONE => ""
                                        | SOME (Note (s)) => s

val intSetToString = fn (ms) =>
      (* 'o' used for function composition (right-to-left) *)
      (String.concat o List.rev o (fn l => "}"::l) o #2)
      (* Argument to result of function composition. *)
      (List.foldl (fn (m,(b,acc)) =>
                    (true,
                     (Int.toString m)::
                     (if b then ","::acc else acc)))
                  (false,["{"])
                  ms)

(* Create a string for an interpretation. *)
val interpToString = fn (i) =>
      let
        (* Here we're binding variables to field values. *)
         val {IntField = n, IntSet = ms} = i
      in
         String.concat ["(",
                        Int.toString n,
                        ", ",
                        intSetToString ms,
                        ")"]
      end

(* Use Interp.rvalue to obtain 'raw' interpretation, then invoke
 * interpToString *)
val hcInterpToString = fn (i) =>
      interpToString( Interp.rvalue i )
```

Figure 1.4: firstExampleExternalFns.sml: Part 1 (Output)

```
(* DECISION FUNCTIONS *)
(* Define a text note, which must be 'SOME' *)
val note = fn (s) => SOME(Note(s))

(* Use with 'Munge' operation to define initial interpretation *set* *)
val initInterp = fn i =>
        let
        in
            ( note("Munge"), [ ( note("Munge"), Interp.rhcons { IntField = 8, IntSet =
                            [] }) ] )
        end


val decA = fn i =>
      let
         (* Returns the value: in the original example, a pair (n,ms) *)
         val { IntField = n, IntSet = ms } = i
         (* Syntax: define function to map, followed by list to map over. *)
         val r = List.map (fn j =>
                           let
                              val msU =
                                 List.foldl (fn (k,msU) => k::msU )
                                    []
                                    (List.tabulate (j, fn k => k))
                              val i' =  { IntField = n, IntSet = ms @ msU }
                           in
                              (note("decA"), i')
                           end)
                           (* Set over which to map:
                           *  a list of integers, with j being the identity
                           * fn; maps from 0...n-1 → f(0) ... f(n-1) *)
                          (List.tabulate (n, fn j => j))
      in
         (note("decA"), r)
      end

val decB = fn i =>
      let
         val { IntField = n, IntSet = ms } = i
         val r = List.map (fn j =>
                           let
                              val msD = List.filter (fn k => k mod j = 0) ms
                              val i' = { IntField = n,
                                IntSet = List.filter
                                     (fn el =>
                                          not (List.exists
                                                 (fn e' => e' = el)
                                                 msD))
                                      ms }
                           in
                              (note("decB"), i')
                           end)
                          (if n > 0
                              then List.tabulate (n - 1, fn j => j + 1)
                           else [])
      in
         (note("decB"), r)
      end


val decC = fn i =>
      let
         val { IntField = n, IntSet = ms } = i
         val r = List.exists (fn e => e = (n div 2)) ms
      in
         (note("decC"), r)
      end
```

Figure 1.5: firstExampleExternalFns.sml: Part 2 (Decisions)

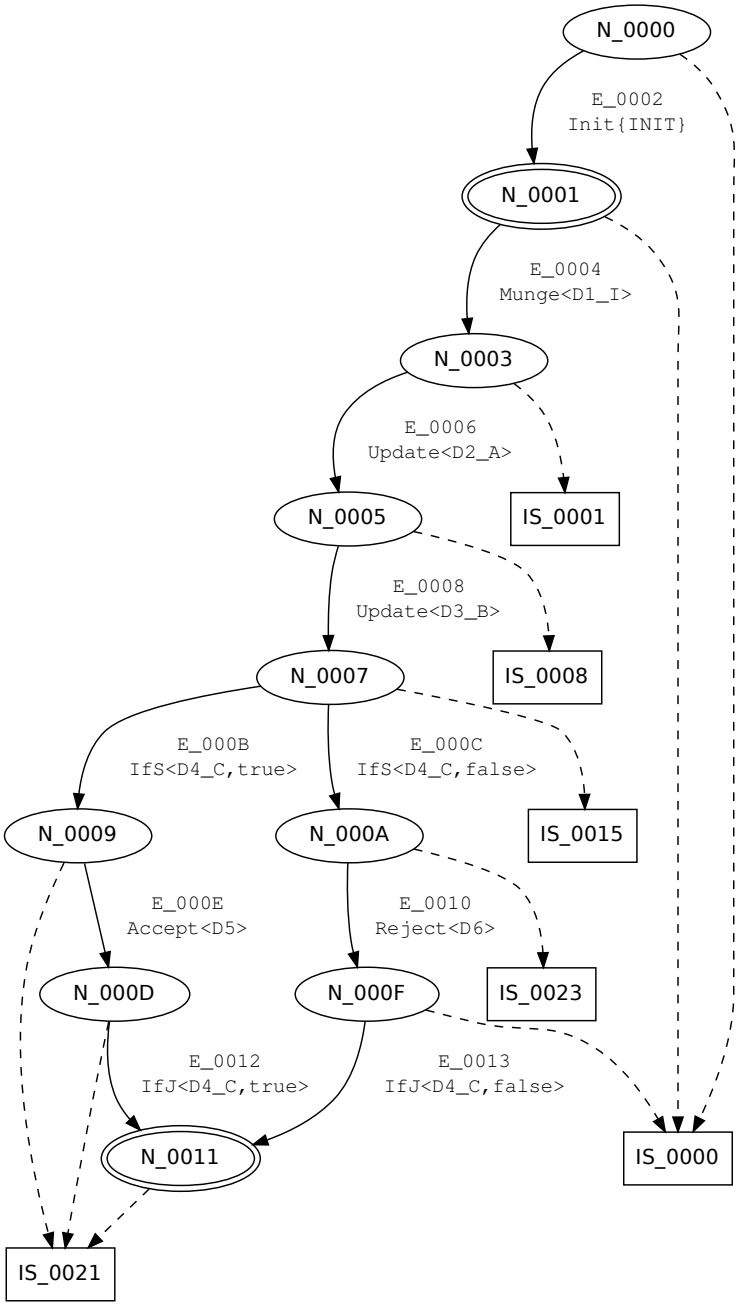


Figure 1.6: RSL Trace for execution of `firstExample.rsl`

```
Node → InterpSet::
N_0000: IS_0000
N_0001: IS_0000
N_0003: IS_0001
N_0005: IS_0008
N_0007: IS_0015
N_0009: IS_0021
N_000A: IS_0023
N_000D: IS_0021
N_000F: IS_0000
N_0011: IS_0021


InterpSet::
IS_0000: {}
IS_0001: {I_0000}
IS_0008: {I_0000, I_0001, I_0002, I_0003, I_0004, I_0005, I_0006, I_0007}
IS_0015: {I_0000, I_0008, I_0009, I_000A, I_000B, I_000C, I_000D, I_000E, I_000F, I_0010, I_0011,
     I_0012, I_0013, I_0014}
IS_0021: {I_0009, I_000B, I_000C, I_000D, I_000F, I_0011}
IS_0023: {I_0000, I_0008, I_000A, I_000E, I_0010, I_0012, I_0013, I_0014}


Interp::
I_0000: (8, {})
I_0001: (8, {0})
I_0002: (8, {0,1})
I_0003: (8, {0,1,2})
I_0004: (8, {0,1,2,3})
I_0005: (8, {0,1,2,3,4})
I_0006: (8, {0,1,2,3,4,5})
I_0007: (8, {0,1,2,3,4,5,6})
I_0008: (8, {5,3,1})
I_0009: (8, {5,4,2,1})
I_000A: (8, {6,5,3,2,1})
I_000B: (8, {6,4,3,2,1})
I_000C: (8, {5,4,3,2,1})
I_000D: (8, {6,5,4,3,2,1})
I_000E: (8, {5,3,2,1})
I_000F: (8, {4,3,2,1})
I_0010: (8, {3,1})
I_0011: (8, {4,2,1})
I_0012: (8, {3,2,1})
I_0013: (8, {2,1})
I_0014: (8, {1})
```

Figure 1.7: firstExample.data: output from execution of `firstExample`

Chapter 2

Data Model

The central data structures in an RSL program are the current set of interpretations, the history of unique interpretations, and the trace of executed decisions and their effects. We also discuss bindings of local symbols in RSL functions, though these are not stored within the history of an RSL program.

2.1 Interpretations

The most important data structure in RSL is the current set of *interpretations* that are modified as an RSL program executes. Roughly speaking, interpretations may be understood as a record type, or objects with data fields but no methods.

2.1.1 Interpretation Declarations

Below is a simple interpretation declaration.

```
interp:
  bbox: (int * int) * (int * int)
  density: real
```

This defines an interpretation as a record consisting of two fields, the field `bbox`, which contains a pair of integer pairs (x,y coordinates defining a bounding box), and a pixel density (percentage of foreground pixels within the bounding box), given as a floating point number (in ML terminology, a `real`).

The interpretation declaration of an RSL program must precede all other RSL functions. Field types may be any of those shown in Table 2.1.

2.1.2 Hash-Consing, Interpretations, Interpretation Sets, and `program-types.sml`

The central data structure in an RSL program is the current set of interpretations. Using a technique termed hash-consing[1], all interpretation data is stored so that each unique interpretation is stored only once within the data structures of an RSL program. This allows the set of unique interpretations produced to be stored and recovered at the end of an RSL program, and makes it possible to determine when decisions produce the same intepretation(s).

Put simply, hash-consing involves computing a hash value for every part of an interpretation, and then providing a numeric signature (tag) for each unique value of the hash-consed type. *Sets* of interpretations are hash-consed as well. Equivalence for two objects of a hash-consed type can then be tested using their tags, rather than comparing the entire structure of interpretations. This requires some additional overhead for

Table 2.1: Interpretation field types. See the `rsl/trunk/hash-cons` directory. Some types are defined for different precisions (int, real, word)

Primitive Types (ML)	
bool	boolean
int	integer
word	memory word
real	real number (floating point)
char	character
string	string
unit	the empty tuple, () (analagous to `void` in other languages)
Data Structures (Note: element type *precedes* structure type)	
type1 * ... * *typen*	Tuples (pairs (e.g. int * int), triples, 4-tuples supported)
list	List (examples: int list ; (real * string) list)
vector	Dynamic array (example: real vector)
set	Set (example: (int * (int list)) set)
option	Optional value (e.g. `int option` has type NONE or SOME(int))

adding new elements to the hash tables (in order to define tags uniquely), but makes testing the equivalence between two interpretations fast (in the ideal case with few hash-table collisions, $O(1)$).

In RSL hash-consing of interpretations is implemented using ML functors that map user-provided types to hash-consed versions of those types. Hash-consed types are constructed recursively from primitive types, ending with the outer structure type (e.g. the hash-consed interpretation set). Code used to implement hash-consing may be found in the directory `rsl/trunk/hash-cons`.

The RSL parser (`rslp`) defines a structure `RslTypes.Interp` in the file `program-types.sml` for `program.rsl`, which defines the interpretation type, including its hash-consed type information and related functions. To produce the hash-consed type, each field of the interpretation record is given a separate ML structure, that defines a hash-consed version of the user-provided type for the field. In the example above, two structures for the fields `bbox` and `density` are generated, `RslTypes.Interp.bbox_S` and `RslTypes.Interp.density_S`, where the types for each are hash-consed. The generated code for `RslTypes.Interp.t` defining the hash-consed interpretation record type would look similar to:

```
...
type t = {
  bbox_S : MkHashConsedTuple2(
                      MkHashConsedTuple2(structure H1 : HashedInt
                         structure H2: HashedInt)
                      MkHashConsedTuple2(structure H1 : HashedInt
                         structure H2: HashedInt)),
  density_S: HashedReal }
...
```

The `_S` suffixes are added to make the fact that the field types are hash-consed ML structures explicit, both in the generated code, and in any code that manipulates hash-consed interpretations.

Individual interpretations as well as interpretation sets are also hash-consed and assigned to a structure in a generated .sml types file (e.g. `program-types.sml`). To understand the generated file and some RSL operations it is important to be aware of this translation to hash-consed types, **but the programmer need not manipulate hash-consed data directly**, for reasons we'll now discuss.

Raw Types, `rvalue`, and `rhcons`

Simultaneously with the creation of hash-consed interpretation and interpretation set types, 'raw' types which correspond to the field types provided by the programmer in the interpretation declaration are generated as part of the `RslTypes.Interp` structure in `program-types.sml`. For the example above, the following will be created as the ML structure `RslTypes.Interp.r`:

```
...
type r = {
  bbox : (int * int) * (int * int),
  density: real }
...
```

This defines a type that is a regular ML record with two fields of the types specified in the RSL program itself. 'Raw' interpretations and interpretation sets (defined as a list of 'raw' interpretations) may be obtained from hash-consed data using the `rvalue` function, and hash-consed versions of 'raw' data may be produced using the `rhcons` (recursive hash-cons) function.

Note that these field names match those given by the programmer. The expectation is that an RSL programmer will work primarily with 'raw' interpretations in their programs.

2.1.3 Types and Functions for Hash-Consed Structures

Each hash-consed structure, including those produced for primitives, data structures, interpretations and even interpretation sets themselves must define the types and functions listed below. Note that some of these are not visible within a generated `program-types.sml` file, as some are defined in the functors used to create hash-consed types; see `rsl/trunk/hash-cons/mk-hash-consed.fun` and other `mk-hash-consed-*.fun` and related signature files (.sig) in the same directory.

1. Raw type (`r`): un-consed type, e.g. as given for interpretation declarations
   (a) raw value (`rvalue`): un-consed value, e.g. of type for interpretation declarations
   (b) recursive hash-consing function (`rhcons`): maps raw type to hash-consed type through recursive bottom-up construction
2. Hash-consed type (`t`)
   (a) hash-consed value (`value`): obtain the value of a hash-consed structure.
   (b) hash function (`hash`)
   (c) hash-consing function (`hcons`): create consed type from component types
   (d) unique identifier (`tag`)
   (e) equality test (`equal`): compare tags for hash-consed objects of the same hash-consed type.

Note: the 'raw' type for an interpretation set is a list of interpretations of 'raw' type.

2.1.4 Observation Specifications

Most RSL operations support the use of *observation specifications*. which specify what fields of the interpretation records are to be visible to the decision function, and if applicable, which field(s) are updated by the decision function. For operations that update interpretations, the syntax is:

```
[field1], ..., [fieldn] observing [field_first], ..., [field_last]
```

where field1 ... fieldn are the fields to update, and field_first ... field_last are the fields to additionally observe to determine the updates. All the named fields will be de-consed (that is to say, put in 'raw' format) before being passed to the associated decision function. **Note** that `munge` does not support observation specifications, as the operation is designed to manipulate hash-consed interpretations directly.

For control flow operations (`if, while`), the syntax is:

```
observing [field_first], ..., [field_last]
```

where here we only observe interpretations, as the boolean tests do not manipulate interpretations.

Operations that support observation specifications are indicated in Tables 3.1, 3.2, 3.3 and 3.4.

Observation specifications have two purposes: first, they are used for automatic un-consing of interpretations, and to identify those fields that may be updated by a decision. By providing an explicit observation specification, those fields identified will be un-consed, vs. the default where all fields are un-consed for observing using `Interp.rvalue`. Similarly, for a number of operations the field(s) to update may be idenfitied in an RSL operation, allowing a decision function to return records containing only the idenfitied fields, rather than complete interpretations.

Generally speaking, it is a good idea to explicitly state fields to observe and modify, especially if only one or two are needed, but interpretations contain many fields. This reduces the amount of un-consing that needs to occur when passing interpretations to a decision function, and re-hash-consing when updating modified interpretations. In addition, this assists makes dependencies between interpretation components explicit in the RSL program (note that it is possible for a programmer to observe or not really manipulate fields identified in an observation specification).

Examples: see `rsl/trunk/tests/obs_spec` for simple examples of observation specifications, and associated decision functions.

2.2 Annotations

Preceding the `interp` section, there may be an optional `notes` section of an RSL program, which defines the set of legal annotations returned by decision functions. Annotations store information that may be useful at the end of the program, but will not used by the algorithms interpreting the input data. The declaration syntax is nearly identical to that for interpretations; here is an example.

```
notes:
  Prob: real
  IntensityHistogram: int vector
  (* DEFAULT: Note: string *)
```

Here we have defined a probability annotation `Prob`, and an intensity histogram for greyscale images `IntensityHistogram`, as a vector of integers, using one bin for each possible color (e.g. 0-255). The `Note: string` type is always defined by default, and is only shown above for illustration.

Any of the types shown in Table 2.1 may also be used to define annotation types. Decision functions in RSL may always return NONE (no annotation) in lieu of an annotation value. Defined annotations need to be defined using a constructor, e.g. using Note("My annotation") to construct an annotation of type Note.[1]

Annotations will be discussed further in the context of decision functions and manipulating the RSL history.

[1] in the implementation, all annotation types are defined as part of a single datatype `note`, with one constructor corresponding to each annotation type provided by the programmer. The set of possible annotation types, `RslTypes.Annot.t`, is then defined as a `note option`.

2.3 RSL History

There are three types of data stored in the execution history: a record of unique interpretations and interpretation sets, a record of which decisions were executed and what their inputs and outputs were, and a record of decisions produced at decision points in the trace. These three different aspects of the history are stored using execution trace graphs, with nodes representing decision points with an associated current interpretation set, and edges representing decisions made. A decision record is represented in the attributes of an edge. Additionally, some hash tables are constructed, as described below.

The execution trace graph is currently output as a .dot file, after completion of the main() and/or report() functions.

During execution of an RSL program, the different history components are recorded as listed below. History data is stored in a structure (see `rsl/trunk/rsl/mk-rsl-trace.fun`, and the associated signature file `rsl-trace.sig` in the same directory). This trace is passed to external functions when using 'historical RSL operations,' which we describe later.

1. **The trace graph**, including the data structures to record all nodes (Trace.nodes), all edges (Trace.edges), the initial node (Trace.init), and final node (Trace.fini). The functions Trace.inEdges and Trace.outEdges may be used to obtain incoming and outgoing edges for each node in the trace graph.

2. **Interpretations.** Trace.interpSetAtNode returns the interpretation set at a given node for a trace. Trace.finalInterpSet will return the current set of interpretations at the end of the trace. ***All interpretations may be obtained by folding over all interpretation sets in the trace (see the function allInterpList, defined in mk-rsl-trace.fun).

3. **Interpretation Edge Table.** Trace.getInterpEdgeTable returns a hash table mapping hash-consed interpretations to a list of all edges corresponding to operations that produced the given interpretation. **Annotations for decisions are stored on edges**, including those for individual interpretations or decisions, as well as interpretation sets.

4. **Decision points reached.** Trace.allDecisionTags will return a list of string labels for every decision point reached during the execution of a program.

5. **Decision Point Node Table.** Trace.dpNodeTable returns a hash table mapping strings for decision points to a list of all nodes produced correpsonding to the labeled decision point.

2.3.1 User Annotation Table

In the reporting portion of an RSL program, operations are modified so that in addition to an interpretation or interpretation set being passed to external decisions, RSL makes the current execution trace available to the decision function, along with a hash table from (hash-consed) interpretations to a list of triples of type (string * Annot.t * DGraph.edge.t).

Currently, the `hadd` operation will return a hash table from (hash-consed) interpretations to triples representing edges in the execution trace, and insert these into the current user annotation table. Each entry in the list of triples for an interpretation has a string corresponding to the string label associated with the edge, NONE for the annotation, and a reference to the edge itself. **Note that edges store annotations for decisions, so these can be recovered from the edges.** See `rsl/trunk/rsl/mk-rsl-exec.fun` for the implementation (as AST node type `E_Query`).

This table is manipulated by the hadd operation (which queries the history), and some other historical functions (e.g. hupdate), where the programmer may add annotations of their own.

Note: for reporting functions, the `reject` operation clears both the current set of interpretations **and** empties the current user annotation table.

Example: see rsl/trunk/tests/query, which contains the example program `queryTest1.rsl`. Try running the program using:

```
./queryTest1 testIn.i 4
```

The program will print contents of the user annotation table, and make queries using simple filters.

Note: *operations to query the attribute table have been added (e.g.* `getar`*), so that the programmer does not need to act on the attribute table directly. These functions are defined in the file:*
`rsl/rsl_parser/AstTraceExec.sml`.

2.3.2 Decision Labels

As can be seen in Figure 1.3, RSL operations may have an associated decision label. For example,

```
[ I ]
munge: initInterp
```

Labels the operation `munge: initInterp` with `[ I ]`, where "I" is the label associated with this operation, which adds a single interpretation to the current interpretation set. The `.lrsl` file created by the RSL Parser (`rslp`) contains the list of complete labels used for a program. Every RSL operation other than output statements are numbered in order using `D<num>`: operations not labeled by the programmer will have this unique tag as a decision label, and given decision labels are modified so that `D<num>_` becomes a prefix for the label. In the example above, the generated label for the `munge` operation is `D0_I`.

In the reporting portion of an RSL program, interpretations and associated annotations may be recovered from the RSL history using the `hadd` operation, which uses regular expression matching against decision labels to define the set of decision points from which to add interpretations and associated edges to the current set of interpretations, and the current user annotation table, respectively.

2.3.3 Automatically Generated Decision Label Suffixes

Within the RSL Core library (see `rsl/trunk/rsl/mk-rsl-exec.fun`), control flow operations are given some annotations to provide a finer-grained set of decision labels to match when querying the history. These are summarized in the table below.

Operation	Suffix	Description
`if`	`-true`	Set of interpretations for which the boolean test is true.
	`-false`	Set of interpretations for which the boolean test is false.
	`-all`	Set of interpretations produced from the union of interpretations from true and false branches.
`if all`	`-in`	Set of interpretations received as input. Tags above also defined for `if` statement that is executed after the initial currying of the decision function using the current set of interpretations.
`while`	`-true, -false, -out`	while is implemented using 'if': multiple invocations can lead to multiple nodes in the execution trace (if decision points) having the same label (e.g. `-true`). **See note below**
`while all`	`-in`	Set of interpretations received as input.
`duplicate`	`-out`	Union of interpretation sets produced by all branches.

Table 2.2: Suffixes added to decision points during RSL execution

Note: `accept` operations may be labeled, which can be useful for simply capturing interpretation sets. **For while operations, this must be used to capture the final set of interpretations produced**, as there may be multiple '-out' results produced, due to the current implementation of `while` using `if` statements.

Chapter 3

RSL Syntax and Operations

3.1 RSL Functions

The syntax of RSL functions is quite simple. Here is an example of a regular RSL function, that manipulate the current set of interpretations.

```
fn main ( arg1 ) {
  (* Declarations *)
  i = 5
  message = "a string"
  k = Int.toString(arg1)

  (* RSL operations )
  print "Hello World."
  print "I was given:"
  print k
  [ Out ] accept
}
```

Arguments are provided as names, without types. Also, note that value bindings must precede the RSL operations: expressions for values may be provided as any primitive ML value or single function call (which may be higher-order, i.e. arguments to this function may themselves be functions). More sophisticated bindings and operations may also be produced through embedding ML directly into the RSL function (see Section 4.2).

For reporting/historical functions, the syntax differs only slightly:

```
hfn report ( arg1 ) {
  (* Declarations *)
  i = 5
  message = "a string"
  k = Int.toString(arg1)

  (* RSL operations )
  print "Hello World."
  print "I was given:"
  print k
```

```
    (* Add interpretations from end of main function above, print with
     * edge/annotation information *)
    hadd [ "Out" ]
    hprint showInterpWithAssociatedEdges
}
```

Here we run code nearly identical to that for `main`, but being a historical function additional operations may be used that make use of the program trace, and the user annotation table (see the previous chapter for an explanation of these).

Note that for `report`, the set of command line arguments is enforced as being identical to that for `main`. This is not the case for any other RSL functions, where the set of arguments is programmer-defined.

3.2 RSL Programs

RSL programs are defined, **in-order**, as:

1. An optional list of included SML files (`inc "file"`)
2. An optional 'notes' declaration for annotation types
3. The 'interp' interpretation type declaration
4. Zero or more RSL regular functions
5. Zero or more reporting/historical RSL functions

In an RSL program, regular RSL functions *must* precede historical reporting functions. Both RSL function types may employ any of the RSL operations shown in Tables 3.1 and 3.2, which manipulate the current set of interpretations, affect control flow observing the current set of interpretations, and produce output (again, looking at the current set of interpretations).

However, the operations shown in Tables 3.3 and 3.4 can only be used in historical/reporting functions.

3.2.1 Two Entry Points: `main` and `report`

The first entry point for an RSL function is the `main` regular RSL function. If provided, the program continues with the `report` historical RSL function, from which the history produced by executing `main` may be queried and analyzed.

3.3 External/Decision Functions

Decisions are made in RSL programs by functions that are normally defined outside of the RSL program itself. We refer to these as 'external functions' or 'decision functions.' All decision functions are implemented as Standard ML functions. Standard ML provides support for executing code from other languages using the Foreign Function Interface (FFI) or (much more crudely) through system calls to executable programs.

In addition to defining how interpretations are to be modified, external functions may be used to define boolean values (e.g. for conditional statements), or to produce strings (e.g. for printing to the standard output, or to a text file).

There are two basic types of decision functions; those that operate on individual interpretations ('regular'), and those that operate on interpretations plus historical information ('historical'). In addition, a number of operations such as `update, hupdate` and `if` allow the decision function to be a curried

function that is first applied to a set of interpretations before returning a function to be applied to individual interpretations. We briefly summarize these different types of decision functions below.

3.3.1 Regular Decision Functions

Other than for `munge`, regular decision functions that do not employ an observation specification or 'all' are of the form: `RslTypes.Interp.r → [output-type]` that is to say, a single interpretation is assumed as input, and the appropriate output (boolean, string, or annotated list of interpretations) needs to be returned by the decision function.

[DRAFT: for time, look at rsl/trunk/rsl/rsl-ast.sig for expected types for different operations].

3.3.2 Historical Decision Functions

For historical decision functions (again, other than `munge` or those using 'all'), the input is a triple, consisting of:

```
(RslTypes.Interp.r * ((string * RslTypes.Annot.t * DGraph.Edge.t) list) HashTable
        * Trace.t) → [output-type]
```

This simply states that the decision function must accept an interpretation, the user annotation table (a hash table of (string, annotation, trace graph edge) triples), and a Trace structure (containing information about the program's execution to the current point in the program), and then return the output type expected by an operation.

3.3.3 'all' functions

keyword Whether for regular or historical operations, the **all** keyword is used to indicate an operation where the decision function must be a curried function. The decision function will first be applied to the set of interpretations, and must return the curried function, along with an annotation (which is stored in the history: see `rsl/trunk/rsl/mk-rsl-exec.fun`).

Example: see `rsl/trunk/tests/cond`.

3.4 RSL Operations

In this section we provide a very brief summary of the operations available in RSL, and their syntax.

3.4.1 Regular Function Operations

Operations that may be used in both regular and historical RSL functions are summarized in Tables 3.1 and 3.2.

3.4.2 Historical RSL Operations

Operations that may **only** be used in historical RSL functions are summarized in Tables 3.3 and 3.4. Note that with the exception of `hadd`, the operations are simply variations of the regular function operations, but where the input to the function is a triple rather than a single interpretation or interpretation set. External functions associated with these 'historical' or 'reporting' operations are passed the current state of a user annotation table (in which edges from the execution trace and/or defined by the programmer are stored), along with the complete execution trace.

Table 3.1: RSL Decision and Output Operations

Interpretations	
accept	Accept (keep) the current set of interpretations (no-op)
reject	Reject the current set of interpretations (produces empty set) (in reporting/historical functions, also empties user annotation table)
add [file_name]	Add interpretations to current set from an ifile
update [obs_spec] : [dec_fn]	update each current interpretation using a decision function.
update **all** [obs_spec]: [dec_fn]	As in previous, but first pass current interpretation set to dec_fn, then apply resulting curried function dec_fn_curry to each interpretation.
munge : [dec_fn]	Replace the current hash-consed interpretation set using dec_fn.
Output: Standard Output and Standard Error	
print [string]	Print string to standard output.
print ifile	Print current interpretations as an ifile to standard output.
print [obs_spec] : [str_fn]	Print the string produced by str_fn for each interpretation.
print **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.
error [string]	Print string to standard error.
error ifile	Print current interpretations as an ifile to standard error.
error [obs_spec] : [str_fn]	Print the string produced by str_fn for each interpretation.
error **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.
File Output (**use write ++ to append to rather than overwrite a file**)	
write [string] to [file_name]	Write string to given file_name.
write ifile to [file_name]	Write current interpretations as an ifile to file_name.
write [obs_spec] : [str_fn]	Write the string produced by str_fn for each interpretation.
write **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.

Table 3.2: RSL Control Flow Statements

fun_name([arg list])	Call an RSL function, passing the current set of interpretations (implcitly) along with the given list of arguments.**Regular RSL functions (`fn`) may not invoke historical/reporting functions (indicated by `hfn`)**
if [obs_spec] [bool_fn] { ... } else { ... }	Simple conditional (else branch optional). Note that interpretations are individually passed down the true or false branch; **branches that receive no interpretations are not executed.**
if **all** [obs_spec] [bool_fn] { ... } else { ... }	As above, but first pass current interpretation set to str_fn, then apply the resulting curried function bool_fn_curry to each interpretation.
while [obs_spec] [bool_fn] { ... }	Apply operations in the given scope to each interpretation making bool_fn true, until no no such interpretation remains.
while **all** [obs_spec] [bool_fn] { ... }	As above, but first pass current interpretation set to bool_fn, then apply the resulting curried function bool_fn_curry to each interpretation.
duplicate { ... } . . . { ... }	Create two or more scopes that will process the current set of interpretations in parallel (independently of one another), and return the union of their final interpretation sets as the final result.

Table 3.3: Historical RSL Decision and Output Operations

Querying the History (for Interpretations and Trace Graph Edges)	
hadd [[dec_label list]] [: interp_bool]] [notes : note_bool]	Add interpretations from the indicated decision points, selected by matching any of a list of regular expressions to portions of decision labels (e.g. ["dec", "init"] to match all labeled points in the `firstExample.rsl` program) **Regular expressions are defined using awk syntax.** Interpretations are filtered using the `interp_bool` function (keeping those making the function true), and edges associated with the set of selected interpretations are filtered using `note_bool`, and added to the user annotation table.
hadd **all** [: bool_value]] [notes : bool_value]	As above, but any filters provided are applied to the set of all interpretations produced.
Interpretations	
hupdate [obs_spec] : [dec_fn]	update each current interpretation and **user annotation table** using a decision function.
hupdate **all** [obs_spec]: [dec_fn]	As in previous, but first pass current interpretation set to dec_fn, then apply resulting curried function dec_fn_curry to each interpretation.
hmunge : [dec_fn]	Replace the current hash-consed interpretation set and **user annotation table** using dec_fn.
Output: Standard Output and Standard Error	
hprint [string]	Print string to standard output.
hprint ifile	Print current interpretations as an ifile to standard output.
hprint [obs_spec] : [str_fn]	Print the string produced by str_fn for each interpretation, **also observing the user annotation table and trace**.
hprint **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.
herror [string]	Print string to standard error.
herror ifile	Print current interpretations as an ifile to standard error.
herror [obs_spec] : [str_fn]	Print the string produced by str_fn for each interpretation **also observing the user annotation table and trace**.
herror **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.
File Output (**use write ++ to append**)	
hwrite [string] to [file_name]	Write string to given file_name.
hwrite ifile to [file_name]	Write current interpretations as an ifile to file_name.
hwrite [obs_spec] : [str_fn]	Write the string produced by str_fn for each interpretation **also observing the user annotation table and trace**.
hwrite **all** [obs_spec] : [str_fn]	As in previous, but first pass current interpretation set to str_fn, then apply the resulting curried function str_fn_curry to each interpretation.

Table 3.4: Historical RSL Control Flow Statements

hif [obs_spec] [bool_fn] { ... } else { ... }	Simple conditional (else branch optional) **but also observing the user annotation table and trace**. Note that interpretations are individually passed down the true or false branch; **branches that receive no interpretations are not executed.**
hif **all** [obs_spec] [bool_fn] { ... } else { ... }	As above, but first pass current interpretation set to str_fn, then apply the resulting curried function bool_fn_curry to each interpretation.
hwhile [obs_spec] [bool_fn] { ... }	Apply operations in the given scope to each interpretation making bool_fn true, until no no such interpretation remains, **but also observing the user annotation table and trace**. .
hwhile **all** [obs_spec] [bool_fn] { ... }	As above, but first pass current interpretation set to bool_fn, then apply the resulting curried function bool_fn_curry to each interpretation.

Chapter 4

Pre-Processing and the RSL Compiler

The current RSL compiler `rslc` is implemented using a combination of TXL and SML (Standard ML).

4.1 `inc`: including additional SML source files

At the top of an RSL program, any number of .sml files may be included using the syntax:

```
inc "externalDecs.sml"
inc "anotherFile.sml"

...
```

4.2 Embedding ML in an RSL program

While it is less-than-ideal style, to support rapid prototyping ML code may be embedded directly within an RSL program, using the syntax `(*ML*) ... (*ML*)`. ML may be embedded in the following parts of an RSL program:

- Before the `inc` include declarations,
- Before the `notes` and `interp` declarations,
- Before a regular or reporting/historical RSL function, or
- In the data declaration section at the top of an RSL function.

Note that RSL functions are mapped to SML functions.

Embedded ML that precedes the include and notes/interpretation type declarations will appear in the file `program-types.sml`. The remaining ML code is translated and included in the main generated `program.sml` file produced by `rslp` for a program named `program.rsl`.

An example is provided in `rsl/trunk/tests/mlembed/mlembed.rsl` (note: the program takes one command line argument).

4.3 Command-Line Arguments

In an RSL program, the number of command-line arguments passed must match the number of command-line arguments named as parameters of the `main()` function. Consider the following program, which will say hello to the passed command-line arguments.

```
interp:
    field: int

fn main( name1, name2 ) {
    print "Hello " ^ name1 ^ " and " ^ name2
}

hfn report() {
   print "And a Historical Hello for " ^ name1 ^ " and " ^ name2
}
```

This program will say hello to the two passed command line arguments **twice**: once for the main program, and once for the reporting portion of the program. Note that **the same command line arguments are passed to the** `report()` **and** `main()` **functions with the same identifier names, even if the** `report` **function has arguments**. This means that the reporting function shown above is transformed so that it is equivalent to:

```
hfn report( arg1, arg 2 ) {
    print "And a Historical Hello for " ^ name1 ^ " and " ^ name2
}
```

Bibliography

[1] FILLIÂTRE, J.-C., AND CONCHON, S. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML* (New York, NY, USA, 2006), ACM, pp. 12–19.