Successor ML Features for MLton

by

Kevin M Bradley

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Matthew Fluet

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2015

The project “Successor ML Features for MLton” by Kevin M Bradley has been examined and approved by the following Examination Committee:

Dr. Matthew Fluet
Assistant Professor
Project Advisor

Abstract

Successor ML Features
for MLton

Kevin M Bradley

Supervising Professor: Dr. Matthew Fluet

This project aims to extend the MLton implementation of Standard ML by adding in new language features. Over time the Standard ML programming community has identified various language features and extensions, known as Successor ML which should prove useful in practice. The goal of this project is to identify and implement a subset of these features in the MLton compiler. This report presents the selected features, their syntax, motivation, and the details behind their implementation. The selected features are comprised of line comments, do declarations, optional pattern bars, disjunctive patterns, record extension & update, record punning, extended literals, optional semicolons, and withtype for signatures.

Contents

List of Figures

Chapter 1

Introduction

Every programming language, including Standard ML can be improved upon. The original definition for Standard ML was published in 1990 [5]. In response to the definition, in 1992, Professor Appel published a critique [1] of Standard ML describing some shortcomings of the language. In 1997, The Definition of Standard ML Revised [4] was published. Following this revised definition, despite there still being room for improvement in Standard ML, the formal language definition has not evolved. Over the years the SML community identified various language features and extensions which became known as Successor ML [8]. Some SML implementations, such as the interpreter HaMLet S [6] have implemented a number of these features despite them not being part of the formal definition.

The goal of this project is to take a subset of these features from Successor ML, and add them to the MLton implementation of SML. The features to be implemented were selected based on being of proper scope for this project, while also proving useful to SML program writers. A further requirement of this project is that the features are backwards compatible with MLton. Following this project, any legacy source code which was previously working, when compiled in MLton in the same manner, should not be affected or broken by any of these new features. This report provides background information on SML, MLton, the compilation process in general, and the compilation process in MLton. Following the background, the various features along with their motivation, syntax, and implementation details are given. The mechanism for enabling the features is then described along with the results and analysis of the usefulness of the features which were implemented. The final chapters describe some possible project extensions and conclude the report.

Chapter 2

Background

2.1 Standard ML (SML)

Standard ML is a functional, strict programming language. A strict language is one in which functions have their parameters evaluated completely before the function itself is called. Standard ML is sstatically typed, and it also has type-inference. SML is described by the MLton site as "a programming language that combines excellent support for rapid prototyping, modularity, and development of large programs, with performance approaching that of C." [3]. Various resources about learning and programming in SML can be found on the StandardML page of the MLton site [3].

2.2 MLton and Compilation

There are various SML compilers and interpreters, some which stick to the formal SML 97 definition, some of which do not. This project only implements these new language features in the MLton SML compiler which does support the full SML 97 definition. MLton itself "is an open-source, whole-program, optimizing Standard ML compiler" [2]. Whole program optimizations are a set of optimizations which can be performed based off of the knowledge obtained from analyzing the entire program as a whole. While this is a very useful feature of MLton which leads to smaller executables as dead-code elimination can be performed aggressively, this is not of particular significance as it relates to this project.

2.2.1 Compilation Overview

In general the compilation process is broken up into two main sections: the front and back ends. In the front-end the lexing, parsing, and static analysis passes are performed. The static analysis is composed of both type-checking and elaboration. Between the front-end and back-end is what is known as the point of no return. This point of no return represents the last chance that the compiler writer can tell the program writer that the input program was written incorrectly. Following the point of no return the compiler is required to produce an executable for the given input program. In the back-end the optimizations are performed along with the code generation of the compiled executable.

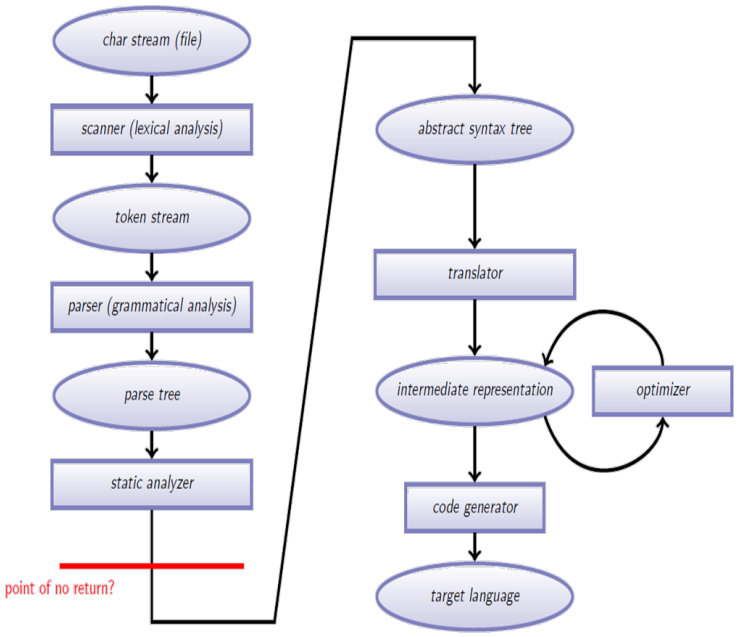

Figure 2.1: The Compilation Process

2.2.2 Compilation Process in MLton

Due to the new features which were implemented being various forms of syntactic sugar, or clearer and more expressive syntax, implementation for these features is done in the front-end stages of compilation in MLton.

Front-End

In the 'front-end' of MLton both the lexing and parsing stages of compilation are performed. The input to this compilation pass is the actual source program which the user had written. The output from this pass is the Abstract Syntax Tree (AST) intermediate language representation of the program. The AST intermediate language describes the program while maintaining source location information to provide better debugging messages. Features which allow for new syntax, for instance new keywords or tokens require making changes to the scanner. The lexer which MLton uses is produced by MLLex, a lexer generator, utilizing the lexical specification file. Features which allow for current syntax to be used in a different manner, for instance allowing optional bars, or do declarations which are described in section 3.2 require changing the parser. MLton's parser is produced by the parser generator MLYacc from the grammar definition file.

Elaboration

This pass performs the role of the static analyzer in the compilation process diagram in Figure 2.1. In this pass both type-inference and type-checking are performed. The program is also defunctorized and elaborated into the next intermediate language representation: CoreML. The CoreML representation is "polymorphic, higher-order, and has nested patterns" [2] which provides an easier means to work with the program further down the compilation process. Features which require new type-checking require making changes to this stage of the compilation process. For instance, since implementing disjunctive patterns means that patterns are allowed to be of a new form, the elaboration must be adjusted to be able to understand how to correctly type-check patterns of this form.

Chapter 3

Implementation

3.1 Line Comments

3.1.1 Feature Overview

Many programming languages have both block comments and line comments. Block comments extend from a designated start point to end point, while line comments extend from the start point only until the end of the designated line. This feature is to implement line comments of the form (*).

3.1.2 Implementation

Since line comments themselves are discarded after lexing, the implemenation of line comments was performed by solely adjusting the lexer. When in the initial state of the lexer, when encountering (*) the lexer transitions to a newly created state of reading line comments. In this new state all tokens that are seen are ignored and lexing continues except for when the End Of Line (EOL) token is encountered. When the EOL token is encountered the lexer transitions back to the initial state and resumes lexing as the current line comment has been completed.

```
(* This is a block comment *)
fun someFun x =
  x + 1 (*) This is a line comment
```

Figure 3.1: Example line comment

```
(*) This is now a line comment
This line is no-longer a comment *)
fun someFun x =
  x + 1
```

Figure 3.2: Example of backwards compatibility issue

3.1.3 Backwards Compatibility Note

The implemenation of this feature does impact backwards compatibility when enabled. In the specific case of a program having a block comment opened followed directly by a close parenthesis this will now be treated as a line comment and not a block comment. This may cause errors to arise as following lines of the comment block will no longer be interpreted as comments.

3.2 Do Declarations

3.2.1 Feature Overview

Evaluating functions for their side effects is a common idiom seen in functional programming languages. In SML this is performed by asserting that the return value of the function is unit, which is discarded. This feature aims to perform the same functionality but in a more clear and concise syntax.

```
(* Examples of discarding the return value *)
val () = print "Hello world.\n"
val _ = print "Hello world.\n"

(* Example do declaration *)
do print "Hello world.\n"
```

Figure 3.3: Example do declaration usage

3.2.2 Implementation

Since this feature is purely syntactic sugar it could be implemented by making changes only to the lexer. In the lexer, when encountering a do declaration, it could be translated to its equivalent value declaration and proceed through the further stages of compilation. The issue with this manner of implementation occurs further down the compilation process during the type-checking phase. Should an error be found, the source code of the offending error would be presented to the programmer in the de-sugared form of what they had written. This results in the user seeing error messages containing code that they did not write. In order to avoid this and present the user with clear error messages referencing the code which they wrote, changing purely the lexer is not sufficient. Implementing do declarations requires making changes to the parser and type-checking stages of compilation. No further changes need to be made to the lexer as no new syntax is introduced, namely 'do' is already a keyword. The parser however must be changed to accept the new syntactic form depicted above. For this to occur, the output of the parser, the Abstract Syntax Tree (AST) needs to be able to represent this new form.

```
datatype decNode =
   Abstype of {body: dec,
               datBind: DatBind.t}
  | DoDec of exp
  | Datatype of DatatypeRhs.t
  | Exception of Eb.t vector
  ...
  | Val of {tyvars: Tyvar.t vector,
            vbs: {exp: exp,
                  pat: Pat.t} vector,
            rvbs: {match: match,
                   pat: Pat.t} vector}
```

Figure 3.4: The AST decNode declaration datatype. The DoDec is the AST representation for do declarations. Val is the structure which would be used for doing the equivalent value declaration (and later discarding the value).

The parser is adjusted to recognize declarations of the form 'do exp', and produce the new corresponding AST representation. The type-checker change is to properly handle and type-check the new AST form. The type-checking phase of compilation for this AST form itself is fairly straight-forward in both of its parts: the type-checking of the do declaration, and the creation of the corresponding output (CoreML representation).

- Type-Checking: to type-check a do declaration simply type-check the expression and verify that the expression is indeed of type unit. In the case of the expression not being of type unit produce a corresponding type-checking error.
- Elaborate: to generate the correct CoreML representation, create the corresponding CoreML representation of the syntactic equivalent value declaration

No further changes need to be made as the do declaration as it has been elaborated the same as a regular value declaration, each further stage of compilation should proceed the same.

3.3 Optional Pattern Bar

3.3.1 Feature Overview

In case analysis of patterns in SML it is required that the first clause does not have a pattern bar before it. This feature makes it optional to place a pattern bar before the first clause. This allows for easier copy and pasting of pattern clauses by creating consistency across all of the clauses and removing a required special case (the first clause). To be consistent this optional pattern bar is implemented in three locations where this idiom occurs: patterns, datatypes, and functions.

3.3.2 Implementation

The presence (or lack thereof) of a pattern bar is not information needed for the type-checker and is not part of the AST. Due to this, implementing the optional pattern bar required changing the parser only. The parser now allows for the first case of a pattern to

```
case t of
 | TInt => []
 | TUnit => []
 | TVar a => [a]
```

Figure 3.5: An optional pattern bar is used at the start of the TInt match

have a starting bar present or not. Regardless of whether the bar is present the resultant AST created is the same.

3.4 Disjunctive Patterns

3.4.1 Feature Overview

This feature allows for being able to utilize the same resultant expression for multiple pattern matches without having to rewrite the expression for each match separately.

```
case x of
   (1 | 2) => ...
 | (3 | 4) => ...
```

Figure 3.6: Example usage of a disjunctive pattern

3.4.2 Implementation

Implementing this feature required making changes to the parsing, type-checking, and match-compiling phases of the compiler. Additionally, disjunctive patterns had to be added to the following intermediate language structures: AST patterns, CoreML patterns, and NestedPats.

Parsing

To parse disjunctive patterns a new nonterminal orpats was created which accepts the above syntax and produces the new AST or-pattern. Originally, disjunctive patterns were to be

implemented at the top level of patterns. However, by allowing for disjunctive patterns at this level, several shift/reduce conflicts were introduced. In order to avoid this problem parsing, disjunctive patterns were implemented at the atomic level instead.

```
pat : patnode (Pat.makeRegion' (patnode, patnodeleft,
   patnoderight))

patnode : pat AS pat (Pat.makeAs (pat1, pat2))
      | pat COLON ty (Pat.Constraint (pat, ty))
      | apats     (Pat.FlatApp (Vector.fromList apats))

apats : apat            ([apat])
      | apat apats      (apat :: apats)

apat : apatnode (Pat.makeRegion' (apatnode, apatnodeleft,
   apatnoderight))

apatnode : longvidNoEqual (Pat.Var {name = longvidNoEqual,
                                    fixop = Fixop.None})
      ...
      | WILD             (Pat.Wild)
      | LPAREN pats RPAREN (Pat.tuple (Vector.fromList pats))
      | LBRACKET pats RBRACKET (Pat.List (Vector.fromList pats))
      | LPAREN pat BAR orpats RPAREN (Pat.Or (Vector.fromList
         (pat :: orpats)))
      | LBRACE RBRACE    (Pat.unit)
      | LBRACE patitems RBRACE
```

Figure 3.7: MLton SML grammar snippet. Note the Pat.Or being created at the atomic level where enclosing parenthesis are required

In order to use a disjunctive pattern, it is now required that it be enclosed in parenthesis. This matches the Standard ML of New Jersey (SML/NJ) [7] implementation of disjunctive patterns.

Type-Checking

Type-checking or patterns requires checking to ensure the three properties below hold true. Should this not be the case, a corresponding error message should be given to the user. After type-checking, the AST or-pattern is elaborated into its CoreML or-pattern equivalent.

1. Variables occurring in one sub-pattern must occur in all of the sub-patterns
2. Variable types must be consistent for all of the variable uses across the sub-patterns
3. The types of each of the sub-patterns must be the same

Flattening

As a preprocessing step to the match-compile phase (following the type-checking and elaboration) the or-patterns are flattened out into their syntactic equivalent. In most cases this process is straightforward, while in the case of an or-pattern being embedded within a tuple-pattern is slightly more complex.

```
(* Matching a tuple with an or-pattern *)
case someTuple of
  (x, (T1 y | T2 y), (T1 z | T2 z)) => ...

(* Flattened form of the or-pattern *)
case someTuple of
    (x, T1 y, T1 z) => ...
  | (x, T1 y, T2 z) => ...
  | (x, T2 y, T1 z) => ...
  | (x, T2 y, T2 z) => ...
```

Figure 3.8: Example of a disjunctive pattern being used on a tuple and the corresponding equivalent flattened form.

Once the or-patterns have been flattened out into their equivalents, compilation can continue as normal. Since the flattening is a pre-processing step for the match-compilation phase, any non-exhaustive cases and/or redundant pattern matches will be reported correctly as the checking for these is performed after the flattening.

3.5 Record Punning

3.5.1 Feature Overview

A common practice in SML is to construct records in which there is a field being assigned a variable which has the same name as the field. Currently, patterns allow for this syntax but not records. This feature allows for the same nice usability of mentioning just the field name (when there is a given variable with the same name).

```
(* Returning a record *)
fn {a, b, c} => {a = a, b = b + 1, c = c}

(* With record punning *)
fn {a, b, c} => {a, b = b + 1, c}
```

Figure 3.9: Example of record punning usage

3.5.2 Implementation

This feature is implemented purely by making changes to the parser. The implementation required adding an additional production to the elabel nonterminal of the parser. Originally, elabels were only allowed to be a field followed by the equals operator followed by an expression to assign to the field. The new production for elabels allows for just a vidNoEqual which represents both the field and the expression (the variable name).

The field and the label which elabels produce can be constructed from this identifier. Since the parser is inserting additional code that was not originally written by the user, the region for the expression (the variable identifier with the same name as the field) is designated to be the same as the region of the field which was written. The drawback of this methodology would come in the instance where the programmer wrote down a field and forgot to write down the equals operator and corresponding expression. In this case, the programmer would see an error message saying that the expression is invalid, as if attempting to perform the pun, when in actuality they were not.

```
elabel : field EQUALOP exp (field,exp)
       | vidNoEqual   (Field.Symbol (Vid.toSymbol vidNoEqual),
                       Exp.makeRegion' (Exp.FlatApp
                       (Vector.new1 (Exp.makeRegion'
                       (Exp.Var {name = (Longvid.short vidNoEqual),
                              fixop = Fixop.None},
                              vidNoEqualleft, vidNoEqualright))),
                        vidNoEqualleft, vidNoEqualright))
```

Figure 3.10: When parsing a record pun the vidNoEqual (the field being punned) is translated to the equivalent form of having been defined as field = expression.

3.6 Record Extension & Record Update

3.6.1 Feature Overview

These features allow for an easier means to create a new (and slightly adjusted) record from a record that has already been created. In record extension a new record can be created by assigning some fields and then assigning the remaining fields via extending a record. Similarly, record update allows for a record to be created from another while updating a field.

```
(* Record with 3 fields *)
val r1 = {a = 1, b = 2, c = 3}

(* Record extension *)
val r2 = {d = false, e = 4, ...=r1}
(* Record update *)
val r3 = {r1 where b=4}
```

Figure 3.11: Example usage of record extension and record update

3.6.2 Implementation Difficulties

Record updating itself is a derived form of record extension. This is the methodology behind the HaMLet S [6] interpreter which has implemented both of these features. Implementing record extension as the basis for implementing record updating is not trivial in itself. The parser must be adjusted to allow records to contain an ellipses as the field identifier. This change to the parser would not be difficult, however the structure of how records are represented at the atom level would need to be adjusted. Previously a record would contain a vector where each entry would be a field, and a type. With record extension this would need to be adjusted to also have each entry in the vector have a boolean identifying whether this field was an ellipses. In addition to the vector another field would be needed to identify whether this record was a flexible one or not. These changes would then need to be propagated throughout the MLton infrastructure as it would need to be adjusted to deal with the differing datatype of records.

```
(* Previous record structure *)
datatype 'a t =
  Tuple of 'a vector
| Record of (Field.t * 'a) vector

(* Purposed new record structure *)
datatype ('a, 'b) t =
  Tuple of 'a vector
| Record of (Field.t * bool * 'a) vector * 'b option
```

Figure 3.12: Previous and proposed atomic record structure

During the propagation of this datatype change it was deemed that it may be more worthwhile to implement some other features. This decision was to avoid the significant time commitment when it was not clear how to then go about type-checking record extensions should the structure changes be successful. Type-checking an extended record would be an extremely difficult task. This difficulty stems mostly from the idea of type-checking an ellipses representing the fields which should be duplicated from the other record. Due

to type-inference it is not clear as to how or when the ellipses could be type-checked. For an extended record to be properly typed, the fields that it defines must not already be defined in the record being extended. The record being extended has to be properly defined and in scope at the time of the record extension. Due to this the field names and all of their types of the record being extended must be known in order to properly type-check the record. This type-checking would then have to interact with the type-inference system as determines tries to resolve the type of the ellipses.

Instead of spending a large amount of time towards attempting to implement a feature which may or may not be successful, instead other features were added in its place. Record updating, being a derived form of record extension, was also not implemented. The additional features added were extended literals, optional semicolons, and withtype for signatures. These additional features were all successfully implemented.

3.7 Extended Literals

3.7.1 Feature Overview

This feature allows for binary literals to be used. Additionally underscores may be used throughout literals to group digits together for convenience. The MLton implementation of SML already supports hexadecimal literals; binary literals are extremely similar.

```
val b = 0b10101
val bw = 0wb1010
val x = 4_327_829
```

Figure 3.13: Example instances of extended literals.

3.7.2 Implementation

Implementation of the extended literals is done solely in the scanner. When a literal is encountered with an initial 0b or 0wb it is then converted to the corresponding int or word

utilizing the StringCVT.BIN format. Once in the proper internal representation, no further changes to the compiler are needed. To scan literals with underscores, simply remove all underscores from the literal before proceeding to create the corresponding data type for the literal. Previously MLton accepted '0wx' as the prefix for hexadecimal words, MLton now accepts the prefix 'w' and 'x' in either order. In the case of binary words, MLton accepts the prefix of '0wb' or '0bw'.

3.8 Optional Semicolon

3.8.1 Feature Overview

Similar to optional pattern bars, in sequences of expressions in SML it is required that the last expression does not have a semicolon following it. This feature makes it optional to place a semicolon after the first expression. This allows for easier copy and pasting of pattern clauses by creating consistency across all of the expressions and removing a required special case (the last expression).

```
if (someBool) then (exp1; exp2; exp3;)
              else (exp1)
```

Figure 3.14: An optional semiclon is used is used after the last expression of the series

3.8.2 Implementation

The presence (or lack thereof) of a semicolon is not information needed for the type-checker and is not part of the AST. Due to this, implementing the optional semicolon required changing the parser only. The parser now allows for the last expression in a series of expressions to also be followed by a semicolon. Regardless of whether the semicolon is present the resultant AST created is the same.

3.9 Withtype for Signatures

3.9.1 Feature Overview

The use of the withtype keyword to define type synonyms is allowed in structures but not in signatures. This feature allows for the functionality to be used in signatures as well.

```
signatures S =
  sig
    datatype a = A1 of int | A2 of int * t
    withtype t = int * a option
  end
```

Figure 3.15: Withtype being used in a signature

3.9.2 Implementation

Implementing withtypes for signatures required making small changes to both the parser and type-checking stages of compilation. In the parser, previously it was enforced that for signatures datatypes had a right-hand-side with no withtypes. In this case adjusting the parser meant simply removing this enforcement, and the associated (and now nolonger needed) rules dealing with the datatypeRhsNoWithtype non-terminal.

Changing the type-checker required adjusting the type-checking and elaboration of DatBinds. Previously, only the datatypes of the DatBind were used in type-checking and elaboration while the withtypes were ignored since there would never be any allowed in signatures (enforced by the parser). The adjustment itself was to then typecheck and elaborate DatBinds in signatures in the same manner in which they are done in structures. In structures the withtypes all have their type elaborated. Once all of the withtypes have been elaborated the environment is then extended with these newly elaborated types.

Chapter 4

Feature Usage and Evaluation

4.1 Enabling the Features

To better promote backwards compatibility all new features are of an opt-in approach. All features are turned off by default and to make use of a new feature it must be specifically enabled via MLBasis Annotations. To enable a feature invoke MLton utilizing the corresponding annotation for the desired feature.

```
$mlton -default-ann 'allowExtendedLiterals true' file.sml
```

Figure 4.1: Turning on the extended literals feature for the given file

Feature	MLBasis Annotation
Line Comments	allowLineComments
Do Declarations	allowDoDecls
Optional Pattern Bar	allowOptBar
Disjunctive Patterns	allowOrPats
Record Punning	allowRecPunning
Extended Literals	allowExtendedLiterals
Optional Semicolon	allowOptSemicolon
Withtype for Signatures	allowSigWithtype

Figure 4.2: Features and their Respective Annotations

4.2 Survey of Usefulness and Results

During the early stages of this project a survey was sent out to the MLton user mailing list to determine the usefulness of the planned features. The users were asked to classify each feature as not helpful, somewhat helpful, or very helpful.

Feature	Not helpful	Somewhat helpful	Very helpful
Line Comments	14	11	3
Do Declarations	19	7	2
Optional Pattern Bar	9	12	7
Disjunctive Patterns	1	11	16
Record Punning	3	12	13
Record Extension	3	9	16
Record Update	0	5	23
Extended Literals	Additional feature, was not surveyed		
Optional Semicolon	Additional feature, was not surveyed		
Withtype for Signatures	Additional feature, was not surveyed		

Figure 4.3: Table of features and their corresponding survey results

The value of the feature implementation was not quite as well received as expected, but not poor. In the case of line comments, and do declarations it is understandable that a significant to majority of the users thought of these features as not helpful. In the case of line comments, many users likely see no benefit to utilizing them opposed to the block comments which they have already been using. Similar logic can be applied to do declarations, where performing the equivalent of discarding by declaring the value to be either a wildcard or unit is not of a significant enough inconvenience. Save these two cases every other feature had more users deem it either somewhat or very helpful over being not helpful. It is clear from the results that the most helpful features are disjunctive patterns, record punning, record extension, and record update where each of these features had few to no responses declaring the feature not helpful.

As previously mentioned record extension and record update were not completely implemented in this project. This coincides with the idea that the more useful a feature is, the

more difficult the implementation. In the extreme case, should there be a purposed feature for a language for an outstanding time that has not been added but known to be very useful in practice, it is likely because the feature would not be trivial to add to the language. It was interesting to see the a fairly significant increase in helpfulness of record updates over record extension, despite being a derived form of it. To this end it may be valuable to re-survey the userbase to see if there is a discrepancy between the usefulness of optional pattern bars opposed to optional semicolons. Overall, the survey results were still quite favorable, with not a single user reporting that all of the new features would not be helpful to them.

Chapter 5

Future Work

In general this project could easily be extended by implementing additional new language features and extensions. The first extension of this project would be further work to complete the implementation of record extension. Following this successful implementation, record updating could then be implemented as a derived form of record extension. Additionally, many more features which are described by Successor-ML [8] or other features already implemented by HaMLet S [6] would be potential additions as well. During the survey of usefulness, participants were also asked if there were any other additional SML features which they would find useful. These suggestions included higher order functors, local functors, and vector patterns. These suggested features could be supplemented with further surveying of the SML community to identify some more potential features which could be implemented.

Besides adding more new features it would also be useful to be able to enable and new features in groups or as a whole, opposed to one at a time. While enabling features individually allows for more granular control over which features to use, a general user may prefer to enable all successor features at once with a designated annotation. Other, smaller, enabling control groups may also prove beneficial. For instance it could often be the case that users which enable optional pattern bars, also always enable the similar feature of optional semicolons.

Chapter 6

Conclusions

This report described a number of previously identified Standard ML language features, their motivation, their syntax and the details behind their implementation. This project successfully implemented a number of these features for MLton. Two of the originally identified features to be implemented (record extension and update) were not completed but replaced by other additional features (extended literals, optional semicolons and withtype for signatures). These features are disabled by default and can optionally be turned on by enabling the corresponding MLBasis Annotation for that feature. The majority of these new features were surveyed and deemed useful by the MLton userbase, making this a successful implementation of Successor ML features in MLton.

Bibliography

[1] Andrew W. Appel. A Critique of Standard ML. *Journal of Functional Programming*, 3:391–429, 1993.

[2] MLton. http://www.mlton.org/.

[3] MLton. http://www.mlton.org/StandardML.

[4] David MacQueen Robin Milner, Mads Tofte. *The Definition of Standard ML Revised*. The MIT Press, Cambridge, MA, USA, 1997.

[5] Robert Harper Robin Milner, Mads Tofte. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, USA, 1990.

[6] Andreas Rossberg. HaMLet S. http://www.mpi-sws.org/ rossberg/hamlet/.

[7] Standard ML of New Jersey. http://www.smlnj.org/.

[8] Successor ML. http://successor-ml.org/.