Abstract Value Constructors

Symbolic Constants for Standard ML

William E. Aitken*
John H. Reppy†

TR 92-1290
June 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853

*This work supported, in part, by the ONR under contract N00014-88-K-0409.

†Supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

Abstract Value Constructors

Symbolic Constants for Standard ML*

William E. Aitken† — Cornell University — aitken@cs.cornell.edu

John H. Reppy‡ — Cornell University§ — jhr@research.att.com

August 23, 1994

Abstract

Standard ML (SML) has been used to implement a wide variety of large systems, such as compilers, theorem provers, graphics libraries, and even operating systems. While SML provides a convenient, high-level notation for programming large applications, it does have certain deficiencies. One such deficiency is the lack of a general mechanism for assigning symbolic names to constant values. In this paper, we present a simple extension of SML that corrects this deficiency in a way that fits naturally with the semantics of SML. Our proposal is a generalization of SML's datatype constructors: we introduce *constants* that generalize nullary datatype constructors (like `nil`), and *templates* that generalize non-nullary datatype constructors (like `::`). Constants are identifiers bound to fixed values, and templates are identifiers bound to structured values with labeled holes. Templates are useful because they allow users to treat the representation of structured data abstractly without having to give up pattern matching.

1 Introduction

Standard ML (SML) is a modern high-level language with both a formal definition ([MTH90, MT91]) and good implementations (e.g., [AM87]). SML is being widely used to implement substantial systems; including compilers, theorem provers, graphics libraries, and even operating systems. Although SML is generally well designed, there are certain language features that are known to be useful for system programming, but are not included in SML. One of the more glaring omissions is the lack of a general purpose mechanism for assigning symbolic names to constant values.

---

*A shorter version of this paper was presented at the ACM SIGPLAN Workshop on ML and its Applications [AR92].

†This work was supported, in part, by the ONR under contract N00014-88-K-0409.

‡This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233.

§Current address: AT&T Bell Laboratories, Murray Hill, NJ, 07974.

The use of symbolic identifiers for constant quantities is an essential tool for writing understandable and maintainable software. Every systems programming language provides support for symbolic constants, from the very powerful `#define` mechanism in the C pre-processor ([KR88]), to the constant declarations of Modula-3 ([Nel91]). For example, an elevator control program that uses the values `0` and `1` to represent direction is harder to read than one that uses the symbolic constants `UP` and `DOWN`. In C we might write:

```
typedef int direction_t;
#define UP 0
#define DOWN 1
```

SML provides no mechanism that allows users to specify names for constant values and to use these names in pattern matching. This problem is significant because pattern matching is the principal mechanism used for case analysis and the decomposition of structured values. The lack of a general symbolic constant mechanism limits pattern matching to *concrete* representations (unlike expressions, which can involve abstract symbolic values). This sharply limits the programmer's ability to define abstract objects, since there is a trade-off between abstraction and programmer convenience.

This paper presents *data templates*, a powerful new mechanism for defining symbolic constants. Templates are a generalization of SML's data constructors, and can be used in both expressions and patterns. Furthermore, they work in the presence of separate compilation and parameterized modules. Our mechanism is type-safe and fits naturally into the semantics of SML ([MTH90]).

Our implementation technology has the added benefit of solving an outstanding problem with datatype representation and abstraction. Furthermore, our semantic framework and implementation technology admit a number of extensions, which make the language and semantics more uniform. Although we set our presentation in the context of SML, our mechanism and techniques are applicable to other languages with datatypes and pattern matching.

In the remainder of the paper, we give examples of the utility of our mechanism, describe the interaction between our mechanism and the module system, explain its semantics, sketch its implementation, and discuss related work.

2 ML datatypes and patterns

The SML datatype mechanism provides a high-level notation for declaring new structured types. For example, the declaration

```
datatype 'a list = nil | :: of ('a * 'a list)
```

defines the standard predefined polymorphic list type. This declaration defines two *data constructors*: `nil` is a nullary constructor representing the empty list, and `::` (which is an infix operator) is the list constructor. The expression `1::2::3::nil` evaluates to an integer list of three elements (SML provides the derived notation `[1, 2, 3]` for this construction). Datatypes can also be used to define enumerations; for example,

```
datatype direction_t = NORTH | SOUTH | EAST | WEST
```

The utility of datatypes is greatly enhanced by the use of *pattern matching* in function definitions to do case analysis and value decomposition. For example, the following function takes a list of strings and returns a list with commas inserted between adjacent elements:

```
fun commas [] = nil
  | commas [s] = [s]
  | commas (s::r) = s :: "," :: (commas r)
```

This function consists of three rules (or equations). The left side of a rule is a pattern, the right side is an expression to be evaluated when the pattern is matched. The first rule in `commas` matches the empty list. The second matches the singleton list and binds its single element to `s`. Theoretically, the third matches any list of one or more elements, but, since the order of equations defines their precedence, it can only match lists of two or more elements; it binds `s` to the head of the list and `r` to the tail. Patterns in SML are *linear*; i.e., a variable may occur at most once in a pattern.

The data constructors defined by datatypes have a dual nature; they are used both to construct values (when they are used in expressions), and to destruct values (when they are used in patterns). We use the term *value constructors* to refer to identifiers with this dual nature. If no confusion with type constructors is possible, we abbreviate this to constructors. Our extension of SML provides another mechanism through which identifiers with this dual nature may be defined. Our mechanism allows the definition of *constants*, which are value constructors that refer to fixed pre-existing values, and the definition of *templates*, which are value constructors that refer to fixed, structured values with named holes. Templates can be thought of as the constructors for families of pre-existing values in the same way that non-nullary data constructors may be thought of as constructors for families of new values.

3 Constants

The use of symbolic names to refer to constant values is an essential tool for the writing of understandable, maintainable software. In SML, `val` bindings allow values to be given symbolic names, but these names cannot be matched against in patterns. This is a serious limitation because pattern matching is the principal mechanism for case analysis and the decomposition of structured values. It is also possible, using the `datatype` declaration, to declare identifiers that may be matched against in patterns, but these are not symbolic names for existing values. While these two mechanisms are adequate for many applications, sometimes what is required is a mechanism that allows identifiers both to be bound to particular values and to be matched against in patterns.

For example, consider the implementation of an X Window System library. The X Window System uses a device independent representation of keyboard keys called *keysyms*. We need to provide users with a symbolic name for each of them. Clearly, users need to be able to pattern match against these names so that they can conveniently write programs that respond to different keystrokes differently. Thus, `val` bindings are not suitable to provide these names. Using a datatype to represent keysyms is also unsuitable. The *wire representation* of a keysym, that is, the representation used by the X server, is a 29 bit integer. If the library uses a datatype to represent keysyms, it must also include a function to convert from wire representations to library representations, and another to convert library representations back into wire representations. It is essential that these functions be mutually inverse. Since there are literally thousands of keysyms defined — even *minimum* English language support requires 512 — these functions present a maintenance nightmare. Furthermore, the use of such huge functions adds significant run-time overhead to the marshaling and unmarshaling of messages. Using our mechanism, we can write definitions such as

```
datatype keysym_t = KEYSYM of int
const KS_a = KEYSYM 97
const KS_b = KEYSYM 98
```

Here the representation of a keysym is the constructor `KEYSYM` wrapped around the wire representation. (This is a standard idiom for creating new types isomorphic to existing ones. Compilers can represent `keysym_t` and `int` identically, and treat the constructor `KEYSYM` as a no-op.) The identifiers `KS_a` and `KS_b` are declared as constants that represent the characters 'a' and 'b'. They can be used both in patterns and in expressions. The identifier `KS_a` stands for the structured value `(KEYSYM 97)`. Note that this representation of keysyms has a further advantage over the datatype representation in that it allows keysyms from different keysets to be defined in different modules, enabling users to import only those keysets they actually need.

This use of our mechanism is reminiscent of the `#define` mechanism of the C language [KR88] (arguably, this is one area in which C provides a higher-level notation than SML). SML has a more general pattern matching facility than the C `switch` statement in that patterns can match structured values (e.g., tuples). Therefore, it is natural to allow symbolic names for structured constant values. For example, a calendar program might include the following definitions:

```
datatype date = DATE of {month : int, day : int}

const CHRISTMAS = DATE{month=12, day=25}

fun getPresents CHRISTMAS = true
  | getPresents _ = false
```

4 Templates

Templates are a natural generalization of symbolic constants to allow labeled holes[1]. They provide a mechanism to define a concise syntax for a collection of similarly structured values. A template is defined by a declaration of the form

const *id trivpat* = *patexp*

where *trivpat* is a pattern that involves only variables and record construction, and *patexp* is a pattern that contains only variables, record construction, special constants and other value constructors. Every *patexp* can be viewed both as a pattern and as an expression (they are the syntactic intersection of patterns and expressions). Every variable appearing in *trivpat* must also appear exactly once in *patexp*. This ensures that it is always possible to map back and forth between instances of a template and its expansion.

For example, the template mechanism allows us to define a template for the days of the month July using the declaration

```
const JULY(x) = DATE{month=7,day=x}
```

which allows expressions such as `JULY(17)` to be used to create values for the days in July. It also allows code such as

```
fun nameOfDay (JULY 4) = "Independence Day"
  | nameOfDay CHRISTMAS = "Christmas"
  | ...
```

in which `JULY` is used as a constructor in a pattern match.

A more substantial and more useful example of templates arises in systems that do term manipulation (such as the Nuprl proof development system, or code optimizers). For the sake of concreteness, we set our example in the context of a generalized λ-calculus.

[1]The term *template* was suggested by Dave MacQueen.

A *term* is either a *variable* or the application of an *operator* to a sequence of *bound terms.* For example, λ and **ap** are operators and $\lambda(x.a)$ is a term with x bound in the sub-term a, and $\mathbf{ap}(a;b)$ is term with sub-terms a and b (but no bound variables). One possible representation of this term language uses a different constructor for each operator:

```
datatype term
  = VAR of var
  | LAMBDA of (var * term)
  | AP of (term * term)
  | PLUS of (term * term)
  | ...
```

This representation has a two major deficiencies. First, functions that are largely independent of the operators, such as computing the free variables of a term or substitution, require many similar cases. For example, the code for computing the free variables of a term is:

```
fun freeVars (VAR v) = [v]
  | freeVars (LAMBDA(x, t)) = setMinus (freeVars t, [x])
  | freeVars (AP(t1, t2)) = setUnion (freeVars t1, freeVars t2)
  | freeVars (PLUS(t1, t2)) = setUnion (freeVars t1, freeVars t2)
  | ...
```

where free variable sets are represented by lists. Second, often it is desirable to make the set of operators extensible (for example, as in the Nuprl system), but if this representation of terms is used, the datatype needs to be changed to extend the operator set, and this requires a complete recompilation of the program. Furthermore, extension of the operator set exacerbates the problem with functions like substitution — the addition of a new operator requires that a new case be added to each such function.

An alternative is to define a regular representation that is extensible. The following datatype is such a representation; the set of terms is extended by treating more strings as valid operator names:

```
datatype operator = OP of string
datatype term
  = VAR of var
  | TERM of operator * ((var list * term) list)
```

With this representation, the free variable function requires only two cases:

```
fun freeVars (VAR v) = [v]
  | freeVars (TERM(_, args)) = let
      fun fvb (bndVars, subTerm) = setMinus (freeVars subTerm, bndVars)
      in
        fold (fn (binding, S) => setUnion (fvb binding, S)) args []
      end
```

Furthermore, this code is independent of any extensions to the term system.

Unfortunately, the syntax of expressions and patterns using this representation is quite ugly. For example, the pattern that matches β-redices (i.e., terms of the form $\mathbf{ap}(\lambda(x.s); t)$ is

```
TERM(OP "AP", [([], TERM(OP "LAMBDA", [([x], s)])), ([], t)])
```

compared to

```
AP(LAMBDA(x, s), t)
```

in the first representation. Moreover, the second representation scheme does not provide the syntactic checking given by the first representation. The expression

```
TERM (OP "LAMBDA", [])
```

is a perfectly acceptable member of the type `term` even though terms formed with the λ operator should always have exactly one subterm in which exactly one variable is newly bound. Lastly, the use of strings to name operators adds the overhead of string comparison to pattern matching. All of these problems are elegantly solved using our mechanism. For example, with the following declarations

```
datatype operator = OP of int
datatype term = VAR of var | TERM of operator * ((var list * term) list)

const LAMBDA_OP = OP 0 and AP_OP = OP 1 and PLUS_OP = OP 2 and ...

const AP(p, q)     = TERM(AP_OP, [([], p), ([], q)])
  and LAMBDA(x, t) = TERM(LAMBDA_OP, [([x], t)])
  and PLUS(a, b)   = TERM(PLUS_OP, [([], a), ([], b)])
```

the pattern for β-redices is again `AP(LAMBDA(x, s), t)`, but the code for computing the free variables remains unchanged. While the type `term` still includes many unintended values, disciplined use of the templates `AP`, `LAMBDA` and `PLUS` makes it impossible for users to produce these values accidentally, and as described in the next section, the required discipline can be enforced using the module facility. Thus, we obtain both the succinctness and syntactic checking of the first representation and the flexibility and regular structure of the second.

5 Constructors and the module system

The module system is an important feature of SML. Our proposal meshes elegantly with the module system; moreover, the module system, because it allows the separation of specification and implementation, greatly increases the abstraction achievable with our mechanism. In particular, because the module system allows the programmer to limit the externally visible definitions of a structure, it is possible to limit the constructors available to users of the structure. This is particularly important when a type used to represent a class of objects contains extra elements that do not represent any object. For example, the second datatype for terms defined in Section 4 includes many ill-formed elements. A signature of the form

```
sig
  eqtype term
  const AP : term of term * term
  const LAMBDA : term of var * term
  const PLUS : term of term * term
end
```

would ensure that users of this term type only used values corresponding to well-formed terms. Moreover, because data constructors are simply a special kind of value constructor, it is possible to provide an interface to a structure in which they are made available as constructors without having to make the datatype declaration in which they are declared visible. This in turn allows the constructors of a datatype to be selectively exported.

Our proposal adds constructor specifications to the syntax of signatures. A nullary constructor specification has the form

`const` *id* `:` *type*

while a unary constructor specification has the form

`const` *id* `:` *type* `of` *type'*

In this specification, *type'* is the domain and *type* is the range of the constructor. The syntactic distinction between unary and nullary constructors is required because, in patterns, unary constructors must always appear with arguments and nullary constructors may never appear with arguments. The legality of code such as

```
signature SIG =
  sig
    type unknown
    const K : unknown
  end
```

```
functor F (A : SIG) =
  struct
    fun f A.K = 17
      | f _ = 12
  end
```

depends on K being a nullary operator. Thus it is essential that structures such as

```
structure S = struct
  type unknown = int -> int * int
  const K x = (x, 12)
end
```

not be allowed to match SIG.

6 Projections

One of the asymmetries of the design of SML is that one can define a view of a constructor that may only be used in expressions (by binding it to a variable using a val declaration), but cannot define a view that can only be used in patterns. As an example of the utility of such a mechanism, consider the implementation of *points* in a graphics toolkit. We may want to restrict points to some sub-range of the representable values, but still allow users to decompose points using pattern matching. This might be done as follows

```
abstype pt_t = PTREP of (int * int)
with
  exception PtRange
  fun mkPt (a, b) =
    if (a and b are in range)
    then PTREP(a, b)
    else raise PtRange
  proj PT (x,y) = PTREP (x,y)
end
```

where the proj declaration defines an identifier that can be used only in patterns. It is safe to allow wildcards on the right-hand side of such *projection* declarations.

Just as val specifications in signatures are used to export only the value of a constructor defined in a structure, so also proj specifications can be used to export only its pattern matching behavior. For the same reason that it is required for constructors, we must maintain explicit arity information for projections.

7 Semantics

In the appendix, we give a rigorous formal description of the semantics of SML extended with abstract value constructors in the style of [MTH90, MT91]. Obviously, a certain amount of new syntax is introduced, and new semantic rules are required to describe its meaning. In addition, some minor changes need to be made to the semantics to provide the extra function given by our extensions. The substantive changes required are small, but the need to propagate their effects means that many of the semantic rules appearing in [MTH90] must be updated. This section briefly outlines and motivates the changes we made to the semantics of the Definition in adding our mechanism to SML.

Our semantics includes a treatment of exceptions. It does not provide an interpretation of SML's other constructor `ref`. We ignore it because allowing `ref` as a constructor would (of course) allow it to appear in constant and template declarations. Since the semantics of `ref` depends on the store, this would mean that the semantics of all value constructors might potentially depend on the store. Moreover, it is not entirely clear what the semantics of templates defined using `ref` should be. Consider the declaration:

```
const strange = ref 1
```

Should every use of `strange` in an expression result in a new reference cell being allocated, or should all occurrences refer to the same object? We do not view this ambiguity as a deficiency of our proposal, rather we view it as evidence that SML's treatment of `ref` as a constructor is a mistake.

In the Definition, the only constructors available are data constructors, which evaluate to themselves, and exception constructors, which can be looked up in the exception environment. In our proposal, constructors denote arbitrary values; therefore, their semantics must depend on a more general environment. For reasons described below, we actually use two environments to give their semantics. We use the value environment, that is used to give the semantics of variables, to give the semantics of constructors in expressions, and we use a new environment, called the *projection environment*, to give their semantics in patterns.

The semantics of a non-nullary value constructor C is given using a pair of functions (C_π, C_ι): an injection and a projection. The injection is used to construct values in expressions, and the projection is used to perform the data destructuring in pattern matching. This is a very general mechanism, and allows many extensions. For example, if C_π and C_ι are arbitrary ML functions, we have views in the sense of [Wad87]. The injection and projection functions of our mechanism are quite restricted, and can be efficiently compiled. Moreover, we believe that our proposal provides most of the function that programmers actually desire. Non-nullary data constructors and exception constructors can also be treated in this manner. Using ML-like notation, the functions corresponding to the data constructor `TERM` defined above are

$$\begin{aligned}
\texttt{TERM}_\iota &= \texttt{fn x => TERM x} \\
\texttt{TERM}_\pi &= \texttt{fn (TERM x) => x | _ =>}\ \text{FAIL}
\end{aligned}$$

where FAIL is a special value used to denote match failure. Similarly, the functions associated with the identifier `LAMBDA` in the template definition given above are

$$\begin{aligned}
\texttt{LAMBDA}_\iota &= \texttt{fn (x, t) => TERM(LAMBDA_OP, [([x], t)])} \\
\texttt{LAMBDA}_\pi &= \texttt{fn (TERM(LAMBDA_OP, [([x], t)])) => (x, t) | _ =>}\ \text{FAIL}
\end{aligned}$$

Note the appearance of `LAMBDA_OP` in these functions. Since the scope of `LAMBDA_OP` may differ from that of `LAMBDA`, these functions must record the environment in which they were defined. A template declaration, `const` *id trivpat* = *patexp*, executed when the value environment is VE and the projection environment is PE, associates an *injection closure* $(trivpat, patexp, VE)$ with *id* in the value environment, and a *projection closure* $(trivpat, patexp, VE)$ with *id* in the projection environment. This discussion is formalized by the following rule:

$$\frac{\langle (SE, VE, PE, EE) \vdash constbind \Rightarrow VE', PE' \rangle}{\begin{array}{l}(SE, VE, PE, EE) \vdash con\ trivpat = patexp \langle \text{and } constbind \rangle \Rightarrow \\ \quad \{con \mapsto (trivpat, patexp, VE)\}\langle + VE' \rangle, \{con \mapsto (trivpat, patexp, PE)\}\langle + PE' \rangle\end{array}}$$

We choose not to use ordinary closures to emphasize the restricted nature of the functions denoted.

Intuitively, the semantics of constants is straightforward. We associate each constant identifier c with its value v. Nullary data constructors can be treated similarly: they are associated with themselves. So too can nullary exception constructors: the evaluation of an exception declaration generates a new exception name, with which the exception constructor is associated.

In practice, to avoid the need to define equality on structured objects, we do something slightly different. Nullary data and exception constructors are treated just as described: they are bound to their value in both the value and projection environments. Constants are bound to their value in the the value environment, and to a closure consisting of the right side of their declaration and the current projection environment. Now, pattern matching against a constant can be defined in terms of pattern matching against their right hand side. This discussion is formalized by the following rule:

$$\frac{(SE, VE, PE, EE) \vdash patexp \text{ in Exp} \Rightarrow v \qquad \langle (SE, VE, PE, EE) \vdash constbind \Rightarrow VE', PE' \rangle}{\begin{array}{l}(SE, VE, PE, EE) \vdash con = patexp \langle \text{and } constbind \rangle \Rightarrow \\ \quad \{con \mapsto v\}\langle + VE' \rangle, \{con \mapsto (patexp, PE)\}\langle + PE' \rangle\end{array}}$$

The environment must be recorded to ensure that constructors appearing in a constant's expansion are correctly interpreted. Once again, we could use ordinary closures, but choose to use a special closure representation to emphasize the special, restricted nature of the represented functions.

Projection declarations produce a binding in the projection environment, but no binding in the value environment. The binding produced is analogous to that produced by the constructor declaration with the same body. Variable declarations are treated exactly as before, in particular they have no effect on the projection environment.

The most important aspect of the module system semantics is the appropriate definition of signature-structure matching. Informally a signature Σ matches a structure S if it is less specific: that is, if it has fewer components, is less polymorphic, or elides the constant nature of values. For example, the specification

```
val nil : int list
```

matches the standard list constructor `nil` (which has type `'a list`). The specification

```
const nil : int list
```

also matches the constructor `nil`. When a structure is constrained by a signature (e.g., when used as the argument to a functor), it is necessary to *thin* it by removing the unused components. Thinning also involves mapping constructors to values or projections by discarding their projection or injection functions. It is to facilitate these operations that we use separate environments to maintain the associations between identifiers and their values, and between identifiers and their projections, rather than associating injection-projection pairs with constructors in the value environment. The association between constructors and their injections is maintained using the ordinary value environment. This ensures that the appropriate binding are automatically available when a signature exports an injection only view of a constructor.

The value of an identifier appearing in an expression is always determined by lookup in the value environment. This contrasts with the Definition, where value constructors merely evaluate to themselves without reference to the environment and exception constructors are looked up in the exception environment.

We must include a rule in the semantics to give the value of an application $exp_1\, exp_2$ in which the value of exp_1 is an injection closure $(trivpat, patexp, VE)$. Here *trivpat* is treated as a pattern, and matched with the value of exp_2. Because *trivpat* consists only of variables and tupling, this match always succeeds in type correct programs. Matching produces an environment that records the required association between values and the variables of *trivpat*. The term *patexp* is treated as an expression, and evaluated using this environment and the environment VE stored in the closure. The resulting value is the value of the application. This description can be formalized as follows:

$$\frac{E \vdash \mathit{exp} \Rightarrow (\mathit{trivpat}, \mathit{patexp}, VE) \qquad E \vdash \mathit{atexp} \Rightarrow v \\ \{\}, v \vdash \mathit{trivpat} \text{ in Pat} \Rightarrow VE' \qquad VE + VE' \vdash \mathit{patexp} \text{ in Exp} \Rightarrow v'}{E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow v'}$$

Pattern matching proceeds as in the Definition, except in its treatment of constructors, where the projection environment is used to give their meaning, rather than the current approach in which the semantics of exception constructors is given using the exception environment, and in which datatype constructors are given meaning without reference to an environment.

First, we consider the matching of a value v against a nullary constructor C. If C is bound to either a constructor or an exception name in the projection environment — that is, if it is declared by either a datatype or an exception declaration — then v matches C if and only if it is equal to the projection associated with C. This means that these constructors receive the same interpretation as they do in the Definition, although the mechanism used to assign this interpretation is rather different. Otherwise, C is bound to a constant closure (patexp, PE). To match a value against this sort of constructor, we treat *patexp* as a pattern, and match the value against it, using PE to interpret any constructors appearing in *patexp*. The match against the constructor succeeds if and only if this match succeeds. No variable bindings are produced. This description is formalized as follows:

$$\frac{PE' = PE \text{ of } E \qquad PE'(\mathit{longcon}) = (\mathit{patexp}, PE'') \\ PE'', v \vdash \mathit{patexp} \text{ in Pat} \Rightarrow VE'/\text{FAIL}}{E, v \vdash \mathit{longcon} \Rightarrow VE'/\text{FAIL}}$$

Matching a value v against a pattern $C\,p$, where C is a constructor and p is a pattern, also requires that C be looked up in the projection environment. If C is bound to a constructor or an exception name, then the match succeeds if and only if the value is a pair consisting of a tag equal to the exception name or constructor associated with the constructor and a value v' that matches p. The bindings produced are those produced by the match of v' against p. Otherwise, C is bound to a projection closure $(\mathit{trivpat}, \mathit{patexp}, PE)$. The term *patexp* is treated as a pattern, and matched against v. If this match fails, so does the match against $C\,p$. Otherwise, the set of bindings it produces, which includes bindings for every variable of *trivpat*, is used to evaluate *trivpat* (interpreted as an expression). The resulting value v', is matched against p. The match of v against $C\,p$ succeeds if the match of v' against p does, and when successful it produces the same bindings, otherwise, it fails. This informal description of pattern matching against template constructors is made formal by the following rules:

$$\frac{PE' = PE \text{ of } E \qquad PE'(\mathit{longcon}) = (\mathit{trivpat}, \mathit{patexp}, PE'') \\ PE'', v \vdash \mathit{patexp} \text{ in Pat} \Rightarrow VE' \quad VE' \vdash \mathit{trivpat} \text{ in Exp} \Rightarrow v' \quad E, v' \vdash \mathit{atpat} \Rightarrow VE''/\text{FAIL}}{E, v \vdash \mathit{longcon}\ \mathit{atpat} \Rightarrow VE''/\text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \qquad PE'(\mathit{longcon}) = (\mathit{trivpat}, \mathit{patexp}, PE'') \quad PE'', v \vdash \mathit{patexp} \text{ in Pat} \Rightarrow \text{FAIL}}{E, v \vdash \mathit{longcon}\ \mathit{atpat} \Rightarrow \text{FAIL}}$$

In the static semantics, we must show how the newly added declarations assign types to identifiers. We must also refine the definition of structure-signature matching to explicitly maintain the status and arities of constructors.

We intend that constant and template declarations assign types to the constructors similarly to the corresponding variable declarations. That is, we intend that the following pairs of declarations assign the same type to *id*

`val` *id* `=` *patexp*	and	`const` *id* `=` *patexp*
`val` *id* `=` `fn` *trivpat* `=>` *patexp*	and	`const` *id* *trivpat* `=` *patexp*

This requires that we refine the definition of scope of explicit type variables to account for `const` declarations, that we extend the definition of type closure, and that we introduce the appropriate rules to the static semantics.

The definition of signature matching requires information about the status of identifiers, that is whether they are projections, variables or constructors. In the Definition, this information can be deduced implicitly: an identifier is a constructor if and only if it is datatype constructor, in which case it appears in the entry for the datatype in the type environment, or if it is an exception constructor, in which case it appears in the exception environment. All other identifiers are variables. This implicit determination of identifier status is impossible in our setting. Thus, we must maintain the status of variables explicitly in the environment. We do this by maintaining a new environment, the *status environment* that associates identifiers with their statuses. The definition of signature enrichment is refined to account for status information. For one status environment to enrich another, the latter environment must associate fewer identifiers with statuses, and each identifier given a status by the latter environment must be given a less restrictive status than in the former environment. Constructor status is more restrictive than either projection or variable status, neither of which is more restrictive than the other. Exception constructor status is the most restrictive status of all. It is tempting to treat identifier status solely in the static semantics, but doing so is quite messy because various syntactic conditions, in particular, the linearity of patterns, depend on the status of identifiers.

For reasons discussed earlier, we must also explicitly maintain the arity of constructors and projections. We include this arity information along with the status information in the status environment.

8 Implementation

We have built a simple testbed implementation of our proposal. At compile time, template identifiers are bound to (*trivpat*, *patexp*) pairs (the environment component of an injection or projection closure is coded in the representation choices for the constructors). When a known template occurs in a pattern, we inline expand it. Say that `C` is bound to (tp, pe) and we have the pattern `C` p. We symbolically apply tp to p. This yields a substitution on the variables of tp (which are the same as the variables occurring in pe). Applying the substitution to pe yields a new pattern, which replaces `C` p.

For a functor parameterized by a structure with constructors, there are two implementation issues: what is the representation of the abstract constructors and how are they used in patterns. To handle the first question, we use implicit structure members for the injection and projection functions. Note that these only need to be added when the structure is made abstract (i.e., by functor application), and can be generated as part of signature thinning. The injection function is an ordinary function, while the projection is a function that returns either the sub-values or FAIL. When building a decision tree, the compiler treats abstract templates as ordinary constructors, which allows merging of matches against them. This ensures that when `CONST` is abstract, code such as

```
case v
 of CONST([]) => 0
  | CONST(a::b) => a
```

is compiled so that the projection function associated with `CONST` is called only once. A test against an abstract constructor is a call to the projection function. Of course, the use of functions to implement the construction and destruction associated with abstract constructors may result in a certain degradation of performance. Of particular concern is the loss of merging when two abstract constructors have common structure. (For example, the templates `LAMBDA` and `AP` defined earlier share the structure `TERM(OP _, _::_)`. Any value not matching this pattern cannot possibly match either of them). Other costs include the loss of inline tests and the replacement of branch tables with trees of conditionals. Note that these costs are incurred only when constructors are actually abstract, that is, when functors are used. In particular no penalty is incurred when structures, which have transparent signatures, are used rather than functors. Moreover if macro expansion is used in the implementation of functor application (a reasonable thing to do when compiling a production version of a system), then functors reduce to structures and abstract constructors become concrete.

Our implementation of abstract constructors can also be applied to solve an outstanding problem, described in more detail in [App90], with datatype representation and abstraction. In some implementations of SML, the representation of the datatype defined by `datatype d = A | B of t`, depends on the representation of `t`. If the representation of `t` is appropriate, it is possible to represent the value `B` *exp* with the representation of *exp*. Problems arise when `t` is abstract, since its representation is known at the definition of `d` but not elsewhere. Extending our technique to cover abstract datatype constructors that fall into the danger zone solves this problem. Although our solution to the problem incurs a performance penalty, a less speedy program is better than one that does not run correctly.

9 Related work

Wadler's view mechanism ([Wad87]) shares the objective of allowing data abstraction and pattern matching to cohabit. Views were once part of the Haskell definition ([HW88]), but were dropped because of technical difficulties. Conceptually, a view of a type T, is a datatype T' together with a pair of mappings between T and T'. Ostensibly these maps are isomorphisms, but since they are defined by the user, there is no assurance that the types are truly isomorphic. Views and templates differ in several significant ways. The principal difference is that Wadler's views define maps between concrete representations, whereas templates provide abstract views of a single representation. Because views define different types, a given pattern match can involve only one view. In addition, once a view is defined, it is not possible to add additional constructors (even if other representations admit additional objects). Templates, on the other hand, do not suffer these restrictions. The implementation of views uses the user defined maps to convert between representations; thus, pattern matching can incur arbitrarily large performance penalties.[2] In our scheme, most uses of templates incur no run-time cost, and the worst cost associated with their use is that they may force patterns to be matched serially rather than in parallel. Our presentation of the semantics of templates is more detailed than that of views given in [Wad87]; furthermore, we address the semantic and implementation issues related to separate compilation and parameterized modules.

The CAML system ([WAL$^+$]) provides a mechanism for defining new concrete syntax, by specifying a grammar to map quoted phrases to the internal representation of programs. This mechanism could be used to implement our template mechanism, although the implementation details appear non-trivial. Recently, a *quotation mechanism* has been proposed for SML, which allows terms in some object language to be included in expressions ([Sli91]). This provides some of the syntactic convenience of our mechanism, but it provides no help for pattern matching against terms of the object language.

[2] In fact, there is no guarantee that the maps even terminate.

References

[AM87] A. W. Appel and D. B.. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324. Springer-Verlag, September 1987.

[App90] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 4(3):343–380, November 1990.

[AR92] William E. Aitken and John H. Reppy. Abstract data constructors. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, June 1992.

[HW88] Paul Hudak and Philip Wadler. Report on the functional programming language haskell (draft proposed standard). Technical Report YALEU/DCS/RR-666, Yale University, Department of Computer Science, December 1988.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 2nd edition, 1988.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Mass, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[Sli91] Konrad Slind. Varieties of object language embedding in standard ML, 1991. *unpublished*.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, January 1987.

[WAL+] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. *The CAML Reference Manual (Version 2.6)*. Projet Formel, INRIA-ENS.

A Appendix

A.1 Introduction

In this appendix, we give the semantics of our proposed mechanism. This semantics is presented as a modification of the formal semantics of Standard ML published in [MTH90], referred to as "the Definition," and [MT91] referred to as "the Commentary." For reasons discussed in the body of the paper, our semantics assumes a language without references (but with exceptions). In section A.5, we show how references may be partially restored to the language.

A.2 Syntax

A.2.1 Core

Two new reserved words, `const` and `proj` must be added to the language, so must a new class Proj of identifiers, containing those identifiers that denote value projections. We define a number of new phrase classes:

ConstBind	constant declarations
ProjBind	projection declarations
TrivPat	template arguments
TrivPatRow	template arguments
AtPatExp	atomic template expansions
PatExpRow	template expansion rows
PatExp	template expansions

We define these phrase classes, and extend a few phrase classes of the Definition as follows: (The symbol +::= is used to denote extension of existing phrase classes.)

constbind	::=	*con* ⟨*trivpat*⟩ = *patexp*⟨and *constbind*⟩	
projbind	::=	*proj* ⟨*trivpat*⟩ = *pat*⟨and *projbind*⟩	
atpatexp	::=	*scon*	special constant
		⟨op⟩*var*	variable
		⟨op⟩*longcon*	value constructor
		⟨op⟩*longexcon*	exception constructor
		`{`⟨*patexprow*⟩`}`	record
		trivpat : *ty*	typed
		`(`*patexp*`)`	
patexprow	::=	*lab* = *patexp*⟨, *patexprow*⟩	
patexp	::=	*atpatexp*	atomic
		⟨op⟩*longcon atpatexp*	value construction
		⟨op⟩*longexcon atpatexp*	exception construction
		$patexp_1$ *con* $patexp_2$	infixed value construction
		$patexp_1$ *excon* $patexp_2$	infixed exception construction
		patexp : *ty*	
dec	+::=	`const` *constbind*	constant declaration
		`proj` *projbind*	projection declaration
atpat	+::=	⟨op⟩*longproj*	projection

pat	$+::=$	$\langle$op$\rangle$*longproj atpat*	projection
		pat_1 *proj* pat_2	infixed projection
trivpat	$::=$	$\langle$op$\rangle$*var*	variable
		{*trivpatrow*}	record
		trivpat : *ty*	typed
		(*trivpat*)	
trivpatrow	$::=$	*lab* = *trivpat*$\langle$, *trivpatrow*$\rangle$	

Certain new syntactic restrictions are required:

- No template argument or template expansion may contain the same variable twice.
- In a constant declaration *con* = *patexp*, *no* variables may occur in *patexp*, while in a constant declaration *con trivpat* = *patexp*, the set of variables occurring in *trivpat* must be identical to the set of those appearing in *patexp*.
- In a projection declaration *proj* = *pat*, *no* variables may occur in *pat*, while in a projection declaration *proj trivpat* = *pat*, the set of variables occurring in *trivpat* must be identical to the set of those appearing in *pat*.
- No constant or projection declaration may bind the same identifier twice.

A.2.2 Modules

Two new phrase classes are required.

ConstDesc	constant specification
ProjDesc	projection specification

They are defined, and the necessary changes made to the Definition's phrase classes by the following productions.

spec	$+::=$	`const` *constdesc*	constant specification
		`proj` *projdesc*	projection specification
constdesc	$::=$	*con* : *ty*$\langle$`of` *ty*$\rangle$$\langle$`and` *constdesc*$\rangle$	
projdesc	$::=$	*proj* : *ty*$\langle$`of` *ty*$\rangle$$\langle$`and` *projdesc*$\rangle$	

One new syntactic restriction is required.

- No constant description *constdesc* or projection description *projdesc* may describe the same identifier twice.

A.2.3 Derived Forms

Certain new derived forms, analogous to derived forms given by the Definition are introduced.

Template Arguments *trivpat*

$(trivpat_1\,,\cdots,\,trivpat_n)$	$\{1 = trivpat_1\,,\cdots,\,\overline{n} = trivpat_n\}$	$(n \geq 2)$

Template Argument Rows *trivpatrow*

$id\langle : ty\rangle\langle\,,\,trivpatrow\rangle$	$id = id\langle : ty\rangle\langle\,,\,trivpatrow\rangle$

Template Expansions *patexp*

$()$	$\{\}$	
$(patexp_1\,,\cdots,\,patexp_n)$	$\{1 = patexp_1\,,\cdots,\,\overline{n} = patexp_n\}$	$(n \geq 2)$
$[patexp_1\,,\cdots,\,patexp_n]$	$patexp_1$ `::` $\cdots$ `::` $patexp_n$ `:: nil`	$(n \geq 0)$

Template Expansion Rows *patexprow*

$id\langle : ty\rangle\langle\,,\,patexprow\rangle$	$id = id\langle : ty\rangle\langle\,,\,patexprow\rangle$

A.2.4 Identifier Status

The status of an identifier, that is whether it is an exception constructor, an ordinary constructor, a variable, or a projection is determined by the rules of Appendix B of the Commentary and the following new rules. Rule 4 of Appendix B must be modified as below to account for the possibility that an identifier is already declared as a projection. A new status `p` is added for projections.

3′. A `const` declaration (specification) assigns *id* : `c` for each *id* which it declares (specifies).

3″. A `proj` declaration (specification) assigns *id* : `p` for each *id* which it declares (specifies).

4. A pattern *pat* assigns *id* : `v` for every *id* which does not already have `c`, `p` or `e` status and occurs in *pat*.

4′. A template argument *trivpat* (template expansion *patexp*) assigns *id* : `v` for every *id* which does not already have `c`, `p` or `e` status and occurs in *trivpat* (*patexp*).

A.3 Static Semantics

A.3.1 Core

We define a new class of simple semantic objects called Status. We use it to represent the status and arity of identifiers. It has seven members: one for variables, and two for each of constructors, projections and exceptions (one for each possible arity).

$$s \in \mathrm{Status} = \{\mathtt{v}, \mathtt{c0}, \mathtt{c1}, \mathtt{p0}, \mathtt{p1}, \mathtt{e0}, \mathtt{e1}\}$$

This set is ordered by the partial order $\sqsubseteq$, which is defined as the least partial order in which the following hold: $\mathtt{v} \sqsubseteq \mathtt{c0}$, $\mathtt{p0} \sqsubseteq \mathtt{c0}$, $\mathtt{c0} \sqsubseteq \mathtt{e0}$, $\mathtt{v} \sqsubseteq \mathtt{c1}$, $\mathtt{p1} \sqsubseteq \mathtt{c1}$, and $\mathtt{c1} \sqsubseteq \mathtt{e1}$.

Only one new set of compound semantic objects is needed, the status environments StatEnv. These are finite mappings from identifiers to their statuses. We must also modify the definition of a (static) environment to include the status environment.

$$\begin{array}{rcl} XE & \in & \mathrm{StatEnv} = \mathrm{Id} \xrightarrow{\mathrm{fin}} \mathrm{Status} \\ E \text{ or } (SE, TE, VE, EE, XE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv} \times \mathrm{ExConEnv} \times \mathrm{StatEnv} \end{array}$$

Type variables may occur in constant and projection declarations, so we must explain their binding structure. Recall that constant and projection declarations are typed analogously to value declarations. This means that we must ensure that free type variables occurring in constant and projection declarations become bound at the correct point. For this reason, we must extend the definitions of unguarded type variables and type variable scope that appear in Section 4.6 to account for projection and constructor declarations. An occurrence of a type variable α in a value, projection or constant declaration *dec* is unguarded if it is not part of a smaller value, projection or constant declaration contained in *dec*. Since constant and projection declarations never contain declarations, this means that every type variable occurring in a constant or projection declaration is unguarded. A type variable α is scoped at a given value, projection or constant declaration if it appears unguarded in that declaration, but not in any larger value declaration containing that declaration. Associated with each value, projection and constant declaration is a (finite) set of the explicit type variables that are scoped there. We think of these declarations as being implicitly decorated with this set, which we indicate in the static semantics by subscripting the reserved words `const`, `proj` and `val`.

Because we intend the static semantics of constant declarations to be easily explicable in terms of the static semantics of variable declarations, we must be careful in closing an environment VE that arises from a constant declaration. Since no constant declaration may bind the same identifier twice, for each constructor *con* bound by VE there is a unique *con* $\langle$*trivpat*$\rangle$ = *patexp* that binds *con*. If $VE(con) = \tau$, let $\mathrm{Clos}_{C,constbind}\, VE(con) = \forall\alpha^{(k)}.\tau$, where

$$\alpha^{(k)} = \begin{cases} \mathrm{tyvars}\,\tau \setminus \mathrm{tyvars}\,C & \text{if } \textit{patexp} \text{ is a variable or constructor, or } \textit{trivpat} \text{ is present.} \\ \mathrm{apptyvars}\,\tau \setminus \mathrm{tyvars}\,C & \text{otherwise.} \end{cases}$$

Similarly, for projection declarations, each *proj* that is bound by VE is bound by a unique clause *proj* $\langle$*trivpat*$\rangle$ = *pat*. If $VE(proj) = \tau$, let $\mathrm{Clos}_{C,projbind}\, VE(proj) = \forall\alpha^{(k)}.\tau$, where

$$\alpha^{(k)} = \begin{cases} \mathrm{tyvars}\,\tau \setminus \mathrm{tyvars}\,C & \text{if } \textit{pat} \text{ is a variable, constructor, or projection,} \\ & \quad \text{or } \textit{trivpat} \text{ is present.} \\ \mathrm{apptyvars}\,\tau \setminus \mathrm{tyvars}\,C & \text{otherwise.} \end{cases}$$

We adopt the convention that for any $longid = strid_1 . \cdots . strid_k . id$ and any environment E, $E(longid) = (VE'(id), XE'(id))$ where VE' and XE' are defined by

$$\begin{array}{rcl} VE' & = & VE \text{ of } ((SE \text{ of } \cdots ((SE \text{ of } E) strid_1) \cdots) strid_k) \\ XE' & = & XE \text{ of } ((SE \text{ of } \cdots ((SE \text{ of } E) strid_1) \cdots) strid_k). \end{array}$$

In what follows, inference rules that are numbered are to be read as replacements for the corresponding rules in the Definition, and those without numbers are to be read as additional rules. Unless explicitly mentioned, the remaining rules of the Definition should be used without change.

Declarations $\boxed{C \vdash dec \Rightarrow E}$

$$\frac{C + U \vdash valbind \Rightarrow VE, XE \quad VE' = \mathrm{Clos}_{C,valbind}\, VE \quad U \cap \mathrm{tyvars}\, VE' = \emptyset}{C \vdash \mathtt{val}_U\ valbind \Rightarrow (VE', XE) \text{ in Env}} \tag{17}$$

$$\frac{C + U \vdash constbind \Rightarrow VE, XE \quad VE' = \mathrm{Clos}_{C,constbind}\, VE \quad U \cap \mathrm{tyvars}\, VE' = \emptyset}{C \vdash \mathtt{const}_U\ constbind \Rightarrow (VE', XE) \text{ in Env}}$$

$$\frac{C + U \vdash projbind \Rightarrow VE, XE \quad VE' = \mathrm{Clos}_{C,projbind}\, VE \quad U \cap \mathrm{tyvars}\, VE' = \emptyset}{C \vdash \mathtt{proj}_U\ projbind \Rightarrow (VE', XE) \text{ in Env}}$$

$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE, XE \qquad \forall (t, CE) \in \mathrm{Ran}\, TE, t \notin (T \text{ of } C) \\ TE \text{ maximizes equality} \end{array}}{C \vdash \mathtt{datatype}\ datbind \Rightarrow (VE, TE, XE) \text{ in Env}} \tag{19}$$

$$\frac{\begin{array}{cc} C \oplus TE \vdash \mathit{datbind} \Rightarrow VE, TE, XE & \forall (t, CE) \in \mathrm{Ran}\ TE, t \notin (T \text{ of } C) \\ C \oplus (VE, TE, XE) \vdash \mathit{dec} \Rightarrow E & TE \text{ maximizes equality} \end{array}}{C \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\texttt{end} \Rightarrow \mathrm{Abs}(TE, E)} \tag{20}$$

$$\frac{C \vdash \mathit{exbind} \Rightarrow EE, XE \qquad VE = EE}{C \vdash \texttt{exception}\ \mathit{exbind} \Rightarrow (VE, EE, XE) \text{ in Env}} \tag{21}$$

The only changes made in these rules were those required to propagate the status environment. The rule for constant declarations, together with the rule given below for the static semantics of constant bindings ensures that the following pairs of declarations assign the same type to *id* (recall that PatExp $\subseteq$ Exp and TrivPat $\subseteq$ Pat).)

`val` *id* `=` *patexp*	and	`const` *id* `=` *patexp*
`val` *id* `= fn` *trivpat* `=>` *patexp*	and	`const` *id trivpat* `=` *patexp*

Value Bindings $\boxed{C \vdash \mathit{valbind} \Rightarrow VE, XE}$

$$\frac{C \vdash \mathit{pat} \Rightarrow (VE, XE, \tau) \quad C \vdash \mathit{exp} \Rightarrow \tau \quad \langle C \vdash \mathit{valbind} \Rightarrow VE', XE' \rangle}{C \vdash \mathit{pat} \texttt{ = } \mathit{exp} \langle \texttt{ and } \mathit{valbind} \rangle \Rightarrow VE \langle + VE' \rangle, XE \langle + XE' \rangle} \tag{26}$$

$$\frac{C + VE \vdash \mathit{valbind} \Rightarrow VE, XE}{C \vdash \texttt{rec}\ \mathit{valbind} \Rightarrow VE, XE} \tag{27}$$

Once again the only changes are those required to propagate the status environment.

Constant Bindings $\boxed{C \vdash \mathit{constbind} \Rightarrow VE, XE}$

$$\frac{\begin{array}{c} \langle C \vdash \mathit{trivpat} \text{ in Pat} \Rightarrow (VE, XE, \tau') \rangle \quad C \langle +(VE, XE) \rangle \vdash \mathit{patexp} \text{ in Exp} \Rightarrow \tau \\ \langle \langle C \vdash \mathit{constbind} \Rightarrow VE', XE' \rangle \rangle \end{array}}{\begin{array}{l} C \vdash \mathit{con} \langle \mathit{trivpat} \rangle \texttt{ = } \mathit{patexp} \langle \langle \texttt{and } \mathit{constbind} \rangle \rangle \Rightarrow \\ \qquad \{\mathit{con} \mapsto \tau\} \langle + \{\mathit{con} \mapsto \tau' \to \tau\} \rangle \langle \langle + VE' \rangle \rangle, \{\mathit{con} \mapsto \texttt{c0}\} \langle + \{\mathit{con} \mapsto \texttt{c1}\} \rangle \langle \langle + XE' \rangle \rangle \end{array}}$$

This rule formalizes the earlier discussion of the static semantics of `const` declarations. The *trivpat* (if present) is treated as a pattern, and the *patexp* is treated like an expression. Note the assignment of constant status and the appropriate arity (1 if *trivpat* is present, 0 otherwise) to each constant bound.

Projection Bindings $\boxed{C \vdash \mathit{projbind} \Rightarrow VE, XE}$

$$\frac{\begin{array}{c} \langle C \vdash \mathit{trivpat} \text{ in Pat} \Rightarrow (VE, XE, \tau') \rangle \quad C \vdash \mathit{pat} \Rightarrow (VE, XE, \tau) \\ \langle \langle C \vdash \mathit{projbind} \Rightarrow VE', XE' \rangle \rangle \end{array}}{\begin{array}{l} C \vdash \mathit{proj} \langle \mathit{trivpat} \rangle \texttt{ = } \mathit{pat} \langle \langle \texttt{and } \mathit{projbind} \rangle \rangle \Rightarrow \\ \qquad \{\mathit{proj} \mapsto \tau\} \langle + \{\mathit{proj} \mapsto \tau' \to \tau\} \rangle \langle \langle + VE' \rangle \rangle, \{\mathit{proj} \mapsto \texttt{p0}\} \langle + \{\mathit{proj} \mapsto \texttt{p1}\} \rangle \langle \langle + XE' \rangle \rangle \end{array}}$$

Note that the same variable and status environments are produced for both *trivpat* and *pat*. This ensures that their variables are typed identically. Recall that the syntax requires that they contain exactly the same variables. Note the assignment of projection status and the appropriate arity to each declared projection identifier.

Datatype Bindings $\boxed{C \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE}, \mathit{XE}}$

$$\frac{\begin{array}{c}\mathit{tyvarseq} = \alpha^{(k)} \qquad C, \alpha^{(k)}t \vdash \mathit{conbind} \Rightarrow \mathit{CE}, \mathit{XE} \\ \langle C \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE}, \mathit{XE}' \qquad \forall (t', \mathit{CE}) \in \mathrm{Ran}\,\mathit{TE}, t \neq t' \rangle\end{array}}{\begin{array}{l}C \vdash \mathit{tyvarseq}\ \mathit{tycon} = \mathit{conbind} \langle\ \mathtt{and}\ \mathit{datbind} \rangle \Rightarrow \\ \quad \mathrm{Clos}\,\mathit{CE}\langle + \mathit{VE}\rangle, \{\mathit{tycon} \mapsto (t, \mathrm{Clos}\,\mathit{CE})\}\langle + \mathit{TE}\rangle, \mathit{XE}\langle + \mathit{XE}'\rangle\end{array}} \tag{29}$$

This rule propagates the state environment produced by its hypothesis. It is otherwise the same as the corresponding rule in the Definition.

Constructor Bindings $\boxed{C, \tau \vdash \mathit{datbind} \Rightarrow \mathit{CE}, \mathit{XE}}$

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash \mathit{conbind} \Rightarrow \mathit{CE}, \mathit{XE} \rangle\rangle}{\begin{array}{l}C, \tau \vdash \mathit{con}\langle \mathtt{of}\ \mathit{ty}\rangle\langle\langle \mid \mathit{conbind}\rangle\rangle \Rightarrow \\ \quad \{\mathit{con} \mapsto \tau\}\langle\{\mathit{con} \mapsto \tau' \to \tau\}\rangle\langle\langle + \mathit{CE}\rangle\rangle, \{\mathit{con} \mapsto \mathtt{c0}\}\langle\{\mathit{con} \mapsto \mathtt{c1}\}\rangle\langle\langle + \mathit{XE}\rangle\rangle\end{array}} \tag{30}$$

Note the assignment of constructor status, and the appropriate arity to each declared datatype constructor. Otherwise, this rule is the same as the corresponding rule in the Definition.

Exception Bindings $\boxed{C \vdash \mathit{exbind} \Rightarrow \mathit{EE}, \mathit{XE}}$

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau \quad \tau \text{ is imperative} \rangle \qquad \langle\langle C \vdash \mathit{exbind} \Rightarrow \mathit{EE}, \mathit{XE} \rangle\rangle}{\begin{array}{l}C \vdash \mathit{excon}\langle \mathtt{of}\ \mathit{ty}\rangle\langle\langle \mathtt{and}\ \mathit{exbind}\rangle\rangle \Rightarrow \\ \quad \{\mathit{excon} \mapsto \mathtt{exn}\}\langle + \{\mathit{excon} \mapsto \tau \to \mathtt{exn}\}\rangle\langle\langle + \mathit{EE}\rangle\rangle, \{\mathit{excon} \mapsto \mathtt{e0}\}\langle + \{\mathit{excon} \mapsto \mathtt{e1}\}\rangle\langle\langle + \mathit{XE}\rangle\rangle\end{array}} \tag{31}$$

$$\frac{C(\mathit{longexcon}) = (\tau, s) \qquad \langle C \vdash \mathit{exbind} \Rightarrow \mathit{EE}, \mathit{XE} \rangle}{\begin{array}{l}C \vdash \mathit{excon} = \mathit{longexcon}\langle \mathtt{and}\ \mathit{exbind}\rangle \Rightarrow \\ \quad \{\mathit{excon} \mapsto \tau\}\langle + \mathit{EE}\rangle, \{\mathit{excon} \mapsto s\}\langle + \mathit{XE}\rangle\end{array}} \tag{32}$$

Note the assignment of exception status in rule 31. In rule 32, s is either e0 or e1, because exception constructors are declared (specified) only by exception declarations (specifications), and these always assign exception status. Otherwise, these rules are the same as the corresponding rules in the Definition.

Atomic Patterns $\boxed{C \vdash \mathit{atpat} \Rightarrow (\mathit{VE}, \mathit{XE}, \tau)}$

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \{\}, \tau)} \tag{33}$$

$$\frac{}{C \vdash \mathit{scon} \Rightarrow (\{\}, \{\}, \mathrm{type}(\mathit{scon}))} \tag{34}$$

$$\frac{}{C \vdash \mathit{var} \Rightarrow (\{\mathit{var} \mapsto \tau\}, \{\mathit{var} \mapsto \mathtt{v}\}, \tau)} \tag{35}$$

$$\frac{C(\mathit{longcon}) = (\sigma, \mathtt{c0}) \qquad \sigma \succ \tau}{C \vdash \mathit{longcon} \Rightarrow (\{\}, \{\}, \tau)} \tag{36}$$

$$\frac{C(\mathit{longproj}) = (\sigma, \mathtt{p0}) \qquad \sigma \succ \tau}{C \vdash \mathit{longproj} \Rightarrow (\{\}, \{\}, \tau)}$$

$$\frac{C(\mathit{longexcon}) = (\mathtt{exn}, \mathtt{e0})}{C \vdash \mathit{longexcon} \Rightarrow (\{\}, \{\}, \mathtt{exn})} \tag{37}$$

$$\frac{\langle C \vdash patrow \Rightarrow (VE, XE, \varrho)\rangle}{C \vdash \{\langle patrow\rangle\} \Rightarrow (\{\}\langle + VE\rangle, \{\}\langle + XE\rangle, \{\}\langle + \varrho\rangle \text{ in Type})} \tag{38}$$

$$\frac{C \vdash pat \Rightarrow (VE, XE, \tau)}{C \vdash (\,pat\,) \Rightarrow (VE, XE, \tau)} \tag{39}$$

Rule 35 and rule 46 (below) are the only rules concerning patterns that assign status to identifiers. Both only assign variable status. Note how the three rules for constructors, and exceptions constructors use the status to ensure that the identifier really is nullary. The only changes to rules 33, 34, 38 and 39 are those required to propagate the status environment properly.

Pattern Rows $\boxed{C \vdash patrow \Rightarrow (VE, XE, \varrho)}$

$$\frac{}{C \vdash \ldots \Rightarrow (\{\}, \{\}, \varrho)} \tag{40}$$

$$\frac{\begin{array}{c} C \vdash pat \Rightarrow (VE, XE, \tau) \\ \langle C \vdash patrow \Rightarrow (VE', XE', \varrho) \qquad lab \notin \mathrm{Dom}\,\varrho\rangle \end{array}}{C \vdash lab = pat\langle\, , \, patrow\rangle \Rightarrow (VE\langle + VE'\rangle, XE\langle + XE'\rangle, \{lab \mapsto \tau\}\langle + \varrho\rangle)} \tag{41}$$

The only changes to these rules are those required to propagate the status environment properly.

Patterns $\boxed{C \vdash pat \Rightarrow (VE, XE, \tau)}$

$$\frac{C \vdash atpat \Rightarrow (VE, XE, \tau)}{C \vdash atpat \Rightarrow (VE, XE, \tau)} \tag{42}$$

$$\frac{C(longcon) = (\sigma, \mathtt{c1}) \quad \sigma \succ \tau' \to \tau \quad C \vdash atpat \Rightarrow (VE, XE, \tau')}{C \vdash longcon\ atpat \Rightarrow (VE, XE, \tau)} \tag{43}$$

$$\frac{C(longproj) = (\sigma, \mathtt{p1}) \quad \sigma \succ \tau' \to \tau \quad C \vdash atpat \Rightarrow (VE, XE, \tau')}{C \vdash longproj\ atpat \Rightarrow (VE, XE, \tau)}$$

$$\frac{C(longexcon) = (\tau \to \mathtt{exn}, \mathtt{e1}) \qquad C \vdash atpat \Rightarrow (VE, XE, \tau)}{C \vdash longexcon\ atpat \Rightarrow (VE, XE, \mathtt{exn})} \tag{44}$$

$$\frac{C \vdash pat \Rightarrow (VE, XE, \tau) \qquad C \vdash ty \Rightarrow \tau}{C \vdash pat : ty \Rightarrow (VE, XE, \tau)} \tag{45}$$

$$\frac{C \vdash pat \Rightarrow (VE, XE, \tau) \qquad \langle C \vdash ty \Rightarrow \tau\rangle}{C \vdash id\langle : ty\rangle \ \mathtt{as}\ pat \Rightarrow (VE + \{id \mapsto \tau\}, XE + \{id \mapsto \mathrm{v}\}, \tau)} \tag{46}$$

Note how the three rules for constructors, projections, and exception constructors use the status to ensure that the identifier really does accept an argument. Rule 46 has been modified to generate a variable status for the identifier it binds, and to propagate the status information derived from its sub-pattern. Rules 42 and 45 now propagate the status environment but are otherwise unchanged.

A.3.2 Modules

We need to refine the definition of enrichment so that it accounts for the status environment. All that needs to be changed is the definition of environment enrichment, and the only change required there is the addition of a further clause describing the appropriate relationship between the two status environments. An Environment $E_1 = (SE_1, TE_1, VE_1, EE_1, XE_1)$ enriches an environment $E_2 = (SE_2, TE_2, VE_2, EE_2, XE_2)$, written $E_1 \succ E_2$ if:

1. $\mathrm{Dom}\, SE_1 \supseteq \mathrm{Dom}\, SE_2$ and $SE_1(strid) \succ SE_2(strid)$ for each $strid \in \mathrm{Dom}\, SE_2$.
2. $\mathrm{Dom}\, TE_1 \supseteq \mathrm{Dom}\, TE_2$ and $TE_1(tycon) \succ TE_2(tycon)$ for all $tycon \in \mathrm{Dom}\, TE_2$.
3. $\mathrm{Dom}\, VE_1 \supseteq \mathrm{Dom}\, VE_2$ and $VE_1(id) \succ VE_2(id)$ for all $id \in \mathrm{Dom}\, VE_2$.
4. $\mathrm{Dom}\, EE_1 \supseteq \mathrm{Dom}\, EE_2$ and $EE_1(excon) = EE_2(excon)$ for all $excon \in \mathrm{Dom}\, EE_2$.
5. $\mathrm{Dom}\, XE_1 \supseteq \mathrm{Dom}\, XE_2$ and $XE_1(id) \sqsupseteq XE_2(id)$ for all $id \in \mathrm{Dom}\, XE_2$.

Otherwise, the only changes required are in the semantics of specifications: we show how the status of identifiers is inferred and propagated.

Specifications $\boxed{B \vdash spec \Rightarrow E}$

$$\frac{C \text{ of } B \vdash constdesc \Rightarrow VE, XE}{B \vdash \texttt{const}\, constdesc \Rightarrow (\mathrm{Clos}\, VE, XE) \text{ in Env}}$$

$$\frac{C \text{ of } B \vdash projdesc \Rightarrow VE, XE}{B \vdash \texttt{proj}\, projdesc \Rightarrow (\mathrm{Clos}\, VE, XE) \text{ in Env}}$$

$$\frac{C \text{ of } B \vdash datdesc \Rightarrow TE, VE, XE}{B \vdash \texttt{datatype}\, datdesc \Rightarrow (TE, \mathrm{Clos}\, VE, XE) \text{ in Env}} \qquad (73)$$

$$\frac{C \text{ of } B \vdash exdesc \Rightarrow EE, XE}{B \vdash \texttt{exception}\, exdesc \Rightarrow (\{\}, \{\}, EE, EE, XE)} \qquad (74)$$

Value Descriptions $\boxed{C \vdash valdesc \Rightarrow VE, XE}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE, XE \rangle}{C \vdash var \,:\, ty \langle \texttt{and}\ valdesc \rangle \Rightarrow \{var \mapsto \tau\}\langle + VE\rangle, \{var \mapsto \mathtt{v}\}\langle + XE\rangle} \qquad (82)$$

Constant Descriptions $\boxed{C \vdash constdesc \Rightarrow VE, XE}$

$$\frac{C \vdash ty \Rightarrow \tau \quad \langle C \vdash ty' \Rightarrow \tau' \rangle \quad \langle\langle C \vdash constdesc \Rightarrow VE, XE \rangle\rangle}{\begin{array}{l} C \vdash con \,:\, ty \langle \texttt{of}\ ty' \rangle \langle\langle \texttt{and}\ constdesc \rangle\rangle \Rightarrow \\ \qquad \{con \mapsto \tau\}\langle + \{con \mapsto \tau' \to \tau\}\rangle\langle\langle + VE\rangle\rangle, \{con \mapsto \mathtt{c0}\}\langle + \{con \mapsto \mathtt{c1}\}\rangle\langle\langle + XE\rangle\rangle \end{array}}$$

Projection Descriptions $\boxed{C \vdash projdesc \Rightarrow VE, XE}$

$$\frac{C \vdash ty \Rightarrow \tau \quad \langle C \vdash ty' \Rightarrow \tau' \rangle \quad \langle\langle C \vdash projdesc \Rightarrow VE, XE \rangle\rangle}{\begin{array}{l} C \vdash proj \,:\, ty \langle \texttt{of}\ ty' \rangle \langle\langle \texttt{and}\ projdesc \rangle\rangle \Rightarrow \\ \qquad \{proj \mapsto \tau\}\langle + \{proj \mapsto \tau' \to \tau\}\rangle\langle\langle + VE\rangle\rangle, \{proj \mapsto \mathtt{p0}\}\langle + \{proj \mapsto \mathtt{p1}\}\rangle\langle\langle + XE\rangle\rangle \end{array}}$$

Datatype Descriptions $\boxed{C \vdash datdesc \Rightarrow TE, VE, XE}$

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)} \vdash condesc \Rightarrow CE, XE \\ \langle C \vdash datdesc \Rightarrow TE, VE, XE' \rangle \end{array}}{\begin{array}{l} C \vdash tyvarseq\ tycon = condesc \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \qquad \{tycon \mapsto (t, \mathrm{Clos}\, CE)\}\langle + TE\rangle, \mathrm{Clos}\, CE\langle + VE\rangle, XE\langle + XE'\rangle \end{array}} \qquad (84)$$

Constructor Descriptions $\boxed{C, \tau \vdash condesc \Rightarrow CE, XE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash condesc \Rightarrow CE, XE \rangle\rangle}{\begin{array}{l} C, \tau \vdash con\langle \texttt{of}\ ty\rangle\langle\langle \mid condesc\rangle\rangle \Rightarrow \\ \quad \{con \mapsto \tau\}\langle + \{con \mapsto \tau' \to \tau\}\rangle\langle\langle + CE\rangle\rangle, \{con \mapsto \texttt{c0}\}\langle + \{con \mapsto \texttt{c1}\}\rangle\langle\langle + XE\rangle\rangle \end{array}} \tag{85}$$

Exception Descriptions $\boxed{C \vdash exdesc \Rightarrow EE, XE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau \qquad \mathrm{tyvars}(\tau) = \emptyset\rangle \qquad \langle\langle C \vdash exdesc \Rightarrow EE, XE\rangle\rangle}{\begin{array}{l} C \vdash excon\langle \texttt{of}\ ty\rangle\langle\langle \texttt{and}\ exdesc\rangle\rangle \Rightarrow \\ \quad \{excon \mapsto \texttt{exn}\}\langle + \{excon \mapsto \tau \to \texttt{exn}\}\rangle\langle\langle + EE\rangle\rangle, \{excon \mapsto \texttt{e0}\}\langle + \{excon \mapsto \texttt{e1}\}\rangle\langle\langle + XE\rangle\rangle \end{array}} \tag{86}$$

A.4 Dynamic Semantics

A.4.1 Core

The values that expressions may yield are changed. We remove Addr and := because they are required only for the support of references. We also need to add a class ConClosure of values to represent the values of templates. There is no need for a new class to represent the values of symbolic constants, since they are evaluated when they are declared.

We also need to define an environment in which constant identifiers are mapped to their projection value. The possible values include constructors, exception values, and projection values for symbolic constants. The projection values of symbolic constants are represented using two classes, Constant and ProjClosure; the first is used for unparametrized constants, and the second, for templates.

$$\begin{array}{rcl}
v & \in & \mathrm{Val} = \mathrm{SVal} \cup \mathrm{BasVal} \cup \mathrm{Con} \\
 & & \qquad \cup\, (\mathrm{Con} \times \mathrm{Val}) \cup \mathrm{ExVal} \cup \mathrm{Record} \\
 & & \qquad \cup\, \mathrm{Closure} \cup \mathrm{ConClosure} \\
(trivpat, patexp, PE) & \in & \mathrm{ProjClosure} = \mathrm{TrivPat} \times \mathrm{PatExp} \times \mathrm{ProjEnv} \\
(trivpat, patexp, VE) & \in & \mathrm{ConClosure} = \mathrm{TrivPat} \times \mathrm{PatExp} \times \mathrm{VarEnv} \\
(patexp, PE) & \in & \mathrm{Constant} = \mathrm{PatExp} \times \mathrm{ProjEnv} \\
 & & \mathrm{Proj} = \mathrm{Constant} \cup \mathrm{Con} \cup \mathrm{ExVal} \cup \mathrm{ProjClosure} \\
PE & \in & \mathrm{ProjEnv} = (\mathrm{Con} \cup \mathrm{ExCon} \cup \mathrm{Proj}) \stackrel{\mathrm{fin}}{\to} \mathrm{Proj} \\
(SE, VE, PE, EE) \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{VarEnv} \times \mathrm{ProjEnv} \times \mathrm{ExConEnv}
\end{array}$$

Atomic Expressions $\boxed{E \vdash atexp \Rightarrow v/p}$

$$\frac{VE(longvar) = v}{(SE, VE, PE, EE) \vdash longvar \Rightarrow v} \tag{104}$$

$$\frac{VE(longcon) = v}{(SE, VE, PE, EE) \vdash longcon \Rightarrow v} \tag{105}$$

$$\frac{VE(longexcon) = v}{(SE,\,VE,\,PE,\,EE) \vdash longexcon \Rightarrow v} \tag{106}$$

We consistently obtain the value of an identifier by looking it up in the *value* environment. We include all three rules for emphasis, but in fact only rule 105 is changed (while a note in the Definition states that exceptions are to be looked up in the exception environment, the value of every expression must be stored in both the value and exception environments, so this note may be safely ignored).

Expressions $\boxed{E \vdash exp \Rightarrow v/p}$

$$\frac{E \vdash exp \Rightarrow con \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (con, v)} \tag{112}$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (trivpat, patexp,\, VE) \qquad E \vdash atexp \Rightarrow v \\ \{\}, v \vdash trivpat \text{ in Pat} \Rightarrow VE' \qquad VE + VE' \vdash patexp \text{ in Exp} \Rightarrow v' \end{array}}{E \vdash exp\ atexp \Rightarrow v'}$$

Rules 114 and 115, which give the semantics of `ref` and `:=`, need to be eliminated. For the same reason, the restriction that the constant not be `ref` in rule 112 is dropped. The second rule gives the semantics of template application. Note the use of the fact that both TrivPat and PatExp are subsets of Pat $\cap$ Exp to avoid the need for new rules to explain their semantics.

Declarations $\boxed{E \vdash dec \Rightarrow E'/p}$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{val}\ valbind \Rightarrow (\{\}, VE, \{\}, \{\})} \tag{129}$$

$$\frac{E \vdash constbind \Rightarrow VE, PE}{E \vdash \texttt{const}\ constbind \Rightarrow (\{\}, VE, PE, \{\})}$$

$$\frac{E \vdash projbind \Rightarrow PE}{E \vdash \texttt{proj}\ projbind \Rightarrow (\{\}, \{\}, PE, \{\})}$$

$$\frac{\vdash datbind \Rightarrow CE}{E \vdash \texttt{datatype}\ datbind \Rightarrow (\{\}, CE, CE, \{\})}$$

$$\frac{E \vdash exbind \Rightarrow EE}{E \vdash \texttt{exception}\ exbind \Rightarrow (\{\}, EE, EE, EE)} \tag{130}$$

These rules are concerned with ensuring that the projection (and value) environments are correctly maintained. Note that these rules take into account the discussion in section 2.9 of the Commentary.

Constant Bindings $\boxed{E \vdash constbind \Rightarrow VE, PE/p}$

$$\frac{(SE,\,VE,\,PE,\,EE) \vdash patexp \text{ in Exp} \Rightarrow v \qquad \langle (SE,\,VE,\,PE,\,EE) \vdash constbind \Rightarrow VE', PE' \rangle}{\begin{array}{c}(SE,\,VE,\,PE,\,EE) \vdash con = patexp \langle \texttt{and}\ constbind \rangle \Rightarrow \\ \{con \mapsto v\}\langle + VE' \rangle, \{con \mapsto (patexp, PE)\}\langle + PE' \rangle\end{array}}$$

$$\frac{\langle (SE,\,VE,\,PE,\,EE) \vdash constbind \Rightarrow VE', PE' \rangle}{\begin{array}{c}(SE,\,VE,\,PE,\,EE) \vdash con\ trivpat = patexp \langle \texttt{and}\ constbind \rangle \Rightarrow \\ \{con \mapsto (trivpat, patexp,\, VE)\}\langle + VE' \rangle, \{con \mapsto (trivpat, patexp,\, PE)\}\langle + PE' \rangle\end{array}}$$

Two rules describe the semantics of symbolic constant declarations, one for templates and the other for constants. Note that constants are evaluated as soon as they are declared.

Projection Bindings $\boxed{E \vdash projbind \Rightarrow PE/p}$

$$\frac{\langle (SE,\,VE,\,PE,\,EE) \vdash projbind \Rightarrow PE' \rangle}{(SE,\,VE,\,PE,\,EE) \vdash proj = pat\langle \texttt{and}\ projbind\rangle \Rightarrow \{proj \mapsto (pat,\,PE)\}\langle +PE'\rangle}$$

$$\frac{\langle (SE,\,VE,\,PE,\,EE) \vdash projbind \Rightarrow PE' \rangle}{(SE,\,VE,\,PE,\,EE) \vdash proj\ trivpat = pat\langle \texttt{and}\ projbind\rangle \Rightarrow \{proj \mapsto (trivpat, pat,\,PE)\}\langle +PE'\rangle}$$

Two rules describe the semantics of symbolic projection declarations, one for nullary projections and the other for non-nullary projections.

Datatype Bindings $\boxed{\vdash datbind \Rightarrow CE/p}$

$$\frac{\vdash conbind \Rightarrow CE \qquad \langle \vdash datbind \Rightarrow CE' \rangle}{\vdash tyvarseq\ tycon = conbind\langle \texttt{and}\ datbind\rangle \Rightarrow CE\langle +CE'\rangle}$$

This rule, which gives the dynamic semantics of datatype declarations, is required, in essentially this form, even in the semantics of the Definition. It is included here only because it does not appear in the Definition. See section 2.9 of the Commentary, for a discussion of the issues involved.

Constructor Bindings $\boxed{\vdash conbind \Rightarrow CE/p}$

$$\frac{\langle \vdash conbind \Rightarrow CE' \rangle}{\vdash con\langle\langle \texttt{of}\ ty\rangle\rangle\langle \mid conbind\rangle \Rightarrow \{con \mapsto con\}\langle +CE'\rangle}$$

Like the rule that gives the dynamic semantics of datatype declarations, a rule of this form is required in the semantics of the Definition, but not included there. See section 2.9 of the Commentary, for a discussion of the issues involved.

Atomic Patterns $\boxed{E, v \vdash atpat \Rightarrow VE/\mathrm{FAIL}}$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = con \quad v = con}{E, v \vdash longcon \Rightarrow \{\}} \tag{144}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = con \quad v \neq con}{E, v \vdash longcon \Rightarrow \mathrm{FAIL}} \tag{145}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = en \quad v = en}{E, v \vdash longcon \Rightarrow \{\}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = en \quad v \neq en}{E, v \vdash longcon \Rightarrow \mathrm{FAIL}}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longcon) = (patexp, PE'') \\ PE'', v \vdash patexp \text{ in Pat} \Rightarrow VE'/\mathrm{FAIL} \end{array}}{E, v \vdash longcon \Rightarrow VE'/\mathrm{FAIL}} \tag{$*$}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = con \quad v = con}{E, v \vdash longproj \Rightarrow \{\}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = con \quad v \neq con}{E, v \vdash longproj \Rightarrow \mathrm{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = en \quad v = en}{E, v \vdash longproj \Rightarrow \{\}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = en \quad v \neq en}{E, v \vdash longproj \Rightarrow \text{FAIL}}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longproj) = (patexp, PE'') \\ PE'', v \vdash patexp \text{ in Pat} \Rightarrow VE'/\text{FAIL} \end{array}}{E, v \vdash longproj \Rightarrow VE'/\text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longexcon) = en \quad v = en}{E, v \vdash longexcon \Rightarrow \{\}} \tag{146}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longexcon) = en \quad v \neq en}{E, v \vdash longexcon \Rightarrow \text{FAIL}} \tag{147}$$

The first five rules give the only substantive change. The remaining rules echo these rules for projections and exception constructors. We cannot simply use *longid* because variables are treated differently. Note the use of the projection environment in the interpretation of constants. In rule $*$, VE' is always empty, because *patexp* contains no variables.

Patterns $\boxed{E, v \vdash pat \Rightarrow VE/\text{FAIL}}$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longcon) = con \\ v = (con, v') \qquad E, v' \vdash atpat \Rightarrow VE'/\text{FAIL} \end{array}}{E, v \vdash longcon\ atpat \Rightarrow VE'/\text{FAIL}} \tag{154}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = con \quad v \notin \{con\} \times \text{Val}}{E, v \vdash longcon\ atpat \Rightarrow \text{FAIL}} \tag{155}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longcon) = en \\ v = (en, v') \qquad E, v' \vdash atpat \Rightarrow VE'/\text{FAIL} \end{array}}{E, v \vdash longcon\ atpat \Rightarrow VE'/\text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longcon) = en \quad v \notin \{en\} \times \text{Val}}{E, v \vdash longcon\ atpat \Rightarrow \text{FAIL}}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longcon) = (trivpat, patexp, PE'') \\ PE'', v \vdash patexp \text{ in Pat} \Rightarrow VE' \quad VE' \vdash trivpat \text{ in Exp} \Rightarrow v' \quad E, v' \vdash atpat \Rightarrow VE''/\text{FAIL} \end{array}}{E, v \vdash longcon\ atpat \Rightarrow VE''/\text{FAIL}}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longcon) = (trivpat, patexp, PE'') \\ PE'', v \vdash patexp \text{ in Pat} \Rightarrow \text{FAIL} \end{array}}{E, v \vdash longcon\ atpat \Rightarrow \text{FAIL}}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longproj) = con \\ v = (con, v') \qquad E, v' \vdash atpat \Rightarrow VE'/\text{FAIL} \end{array}}{E, v \vdash longproj\ atpat \Rightarrow VE'/\text{FAIL}} \tag{154}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = con \quad v \notin \{con\} \times \text{Val}}{E, v \vdash longproj\ atpat \Rightarrow \text{FAIL}} \tag{155}$$

$$\frac{\begin{array}{c} PE' = PE \text{ of } E \qquad PE'(longproj) = en \\ v = (en, v') \qquad E, v' \vdash atpat \Rightarrow VE'/\text{FAIL} \end{array}}{E, v \vdash longproj\ atpat \Rightarrow VE'/\text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longproj) = en \quad v \notin \{en\} \times \text{Val}}{E, v \vdash longproj\ atpat \Rightarrow \text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \qquad PE'(longproj) = (trivpat, patexp, PE'') \quad PE'', v \vdash patexp \text{ in Pat} \Rightarrow VE' \quad VE' \vdash trivpat \text{ in Exp} \Rightarrow v' \quad E, v' \vdash atpat \Rightarrow VE''/\text{FAIL}}{E, v \vdash longproj\ atpat \Rightarrow VE''/\text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \qquad PE'(longproj) = (trivpat, patexp, PE'') \quad PE'', v \vdash patexp \text{ in Pat} \Rightarrow \text{FAIL}}{E, v \vdash longproj\ atpat \Rightarrow \text{FAIL}}$$

$$\frac{PE' = PE \text{ of } E \qquad PE'(longexcon) = en \quad v = (en, v') \qquad E, v' \vdash atpat \Rightarrow VE'/\text{FAIL}}{E, v \vdash longexcon\ atpat \Rightarrow VE'/\text{FAIL}} \tag{156}$$

$$\frac{PE' = PE \text{ of } E \quad PE'(longexcon) = en \quad v \notin \{en\} \times \text{Val}}{E, v \vdash longexcon\ atpat \Rightarrow \text{FAIL}} \tag{157}$$

Rule 158, which gives the semantics of `ref` when it appears in patterns, must be omitted. The first five rules above give the only other substantive changes. The remaining rules echo these rules for projections and exception constructors. We could have just used *longid* (since the syntax of the language doesn't allow variables to be applied in patterns), but chose not to for the sake of symmetry with the atomic case. Again, note the use of the projection environment in the interpretation of constants.

A.4.2 Modules

The only changes required here are some minor modifications to the way interfaces are defined and built to ensure that structures are cut down appropriately by signatures.

Interfaces now contain an injection (value) and a projection component.

$$(IE, vars, projs,\ excons) \text{ or } I \in \text{Int} = \text{IntEnv} \times \text{Fin(Id)} \times \text{Fin(Id)} \times \text{Fin(ExCon)}$$

Inter and $\downarrow$ are changed to handle projection environments and the projection component of interfaces.

$$\text{Inter}(SE, VE, PE, EE) = (IE, \text{Dom}\, VE, \text{Dom}\, PE, \text{Dom}\, EE)$$

where IE is as before, $IE = \{strid \mapsto \text{Inter}E; SE(strid) = E\}$.

$$(SE, VE, PE, EE) \downarrow (IE, vars, excons, projs) = (SE', VE', PE', EE')$$

where SE' is as before, $SE' = \{strid \mapsto E \downarrow I; SE(strid) = E \text{ and } IE(strid) = I\}$, and

$$\begin{aligned} VE' &= VE \downarrow vars \\ PE' &= PE \downarrow projs \\ EE' &= EE \downarrow excons \end{aligned}$$

with $\downarrow$ now denoting function domain restriction.

We also provide the following semantic rules that ensure that the appropriate identifiers are added to the interface for each specification.

Specifications $\boxed{IB \vdash spec \Rightarrow I}$

$$\frac{\vdash valdesc \Rightarrow vars}{IB \vdash \texttt{val}\ valdesc \Rightarrow (\{\}, vars, \emptyset, \emptyset)} \tag{176}$$

$$\frac{\vdash constdesc \Rightarrow cons}{IB \vdash \texttt{const}\ constdesc \Rightarrow (\{\}, cons, cons, \emptyset)}$$

$$\frac{\vdash \mathit{projdesc} \Rightarrow \mathit{projs}}{\mathit{IB} \vdash \texttt{proj}\,\mathit{projdesc} \Rightarrow (\{\}, \emptyset, \mathit{projs}, \emptyset)}$$

$$\frac{C \vdash \mathit{datdesc} \Rightarrow \mathit{cons}}{\mathit{IB} \vdash \texttt{datatype}\,\mathit{datdesc} \Rightarrow (\{\}, \mathit{cons}, \mathit{cons}, \emptyset)}$$

$$\frac{C \vdash \mathit{exdesc} \Rightarrow \mathit{excons}}{\mathit{IB} \vdash \texttt{exception}\,\mathit{exdesc} \Rightarrow (\{\}, \mathit{excons}, \mathit{excons}, \mathit{excons})} \tag{177}$$

Constant Descriptions $\boxed{\vdash \mathit{constdesc} \Rightarrow \mathit{cons}}$

$$\frac{\langle \vdash \mathit{constdesc} \Rightarrow \mathit{cons} \rangle}{\vdash \mathit{con}\langle \texttt{and}\ \mathit{constdesc} \rangle \Rightarrow \mathit{con}\langle \cup \mathit{cons} \rangle}$$

Projection Descriptions $\boxed{\vdash \mathit{projdesc} \Rightarrow \mathit{projs}}$

$$\frac{\langle \vdash \mathit{projdesc} \Rightarrow \mathit{projs} \rangle}{\vdash \mathit{proj}\langle \texttt{and}\ \mathit{projdesc} \rangle \Rightarrow \mathit{proj}\langle \cup \mathit{projs} \rangle}$$

Datatype Descriptions $\boxed{\vdash \mathit{datdesc} \Rightarrow \mathit{cons}}$

$$\frac{\vdash \mathit{condesc} \Rightarrow \mathit{cons} \qquad \langle \vdash \mathit{datdesc} \Rightarrow \mathit{cons}' \rangle}{\vdash \mathit{condesc}\langle \texttt{and}\ \mathit{datdesc} \rangle \Rightarrow \mathit{cons}\langle \cup \mathit{cons}' \rangle}$$

Constructor Descriptions $\boxed{\vdash \mathit{condesc} \Rightarrow \mathit{cons}}$

$$\frac{\langle \vdash \mathit{condesc} \Rightarrow \mathit{cons} \rangle}{\vdash \mathit{con}\langle \mid \mathit{condesc} \rangle \Rightarrow \mathit{con}\langle \cup \mathit{cons} \rangle}$$

A.5 References

As commented at the beginning of the appendix, we have restricted our attention to a language without references. This is actually a greater restriction than is really needed. It suffices to treat `ref` as a variable rather than a constructor. The required changes to the dynamic semantics presented earlier are quite simple. First, addresses and the special value `:=` must be restored to the set of values. Additionally, two new special values are required, `ref` and `!`. Rules 114 and 115 are restored. A new rule for `!` is also required.

$$\frac{s, E \vdash \mathit{exp} \Rightarrow \texttt{!} \qquad s, E \vdash \mathit{atexp} \Rightarrow a}{s, E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow s(a)}$$

The initial status of `ref` is `v` rather than `c`, the initial value environment, VE'_0, is defined by

$$\mathit{VE}'_0 = \{\mathit{id} \mapsto \mathit{id};\, \mathit{id} \in \mathrm{BasVal}\} + \{\texttt{:=} \mapsto \texttt{:=}\} + \{\texttt{ref} \mapsto \texttt{ref}\} + \{\texttt{!} \mapsto \texttt{!}\}.$$

The Definition's declaration of `!` in terms of `ref`, which is no longer valid, is also removed.