Technical Report HW-MACS-TR-0071

Challenges of a type error slicer for the SML language

Vincent Rahli J. B. Wells Fairouz Kamareddine

ULTRA group, Heriot-Watt University

Abstract

SML (a higher-order function-oriented imperative programming language) depends on automated inference of sophisticated type information. Existing implementations have confusing type error messages. *Type error slicing* gives the programmer more helpful type error information: (1) It identifies all program points that contribute to a type error rather than blaming just one point. (2) It exhibits a slice of the original program's syntax instead of showing an internal representation of some program subtree which may have been substantially transformed. (3) It avoids showing internal details of inferred types which were not written by the programmer. A type error slice contains all information needed to understand the error, and avoids irrelevant information.

Similar to other previous work for better handling of type errors, type error slicing was initially developed for a very tiny toy language: the formal work was for a core barely larger than the λ-calculus, and the implementation had only a few additional primitives. Extending type error slicing to a full programming language is extremely challenging, and to do this for SML we needed a number of innovations. Some issues would be faced for any language, and some are SML-specific but representative of the complexity of language-specific issues likely to be faced by other languages. As an example of a generic issue, the known ways of generating constraints that avoid an explosion in the number of constraints conflict with the need to exclude program parts that are irrelevant to errors, so we developed new ways of handling constraints. As an example of a SML-specific issue, the syntactic class of identifiers in SML depends on earlier declarations, but passing an environment as input makes it more difficult to ensure that error slices cover the right amount of program text, so we developed context sensitive constraints that avoid needing to know identifier syntactic class. Other issues include such things as how to indicate (ir)relevance of various program details, handling also context-sensitive syntax errors, and integration with a program editor. A bonus is that our methods in effect also provide a compositional (bottom up) type inference algorithm for the SML language, which may be useful for reasons other than type error explanation.

1. Introduction

Earlier type inference algorithms for SML. SML is a higher-order function-oriented imperative programming language. SML (and similar languages like OCaml, Haskell, etc.) has polymorphic types allowing considerable flexibility, and almost fully automatic type inference, which frees the programmer from needing to write explicit types. We say "almost fully" because SML has some cases where explicit types are required, e.g., as part of datatype definitions, and type annotations sometimes needed for "flexible" record patterns as in the function `fn {foo, ...} => foo`.

Traditional type inference algorithms take as input an expression and a type environment (and sometimes some other parameters) and output among other things a type of the expression w.r.t. the type system and the (possibly modified) type environment. When the expression is not typable w.r.t. the type system, these algorithms will output an error message. These algorithms have sometimes as a secondary goal efficiency and/or accuracy of the location of the type errors. Our concern is on the latter point.

Milner's W algorithm [4] is the original type-checking algorithm of the purely applicative part (variables, abstractions, applications and polymorphic let expressions) of the ML language. Given an expression E and a type environment Γ covering the free variables of E, if E is typable then W outputs a type σ of E and a substitution sub. The type σ is the principal type of E w.r.t. the application of sub to Γ. If E is not typable, an error is reported. W implementations generally give error messages that are relative to the syntax tree node the algorithm was visiting when unification failed. More precisely, Lee and Yi [14] report that W "fails only at an application expression where its two subexpressions (function and argument) have conflicting types".

Following W, some other algorithms have been developed that try to get better locations by arranging that untypability will be discovered when visiting a different syntax tree node. For example, the folklore algorithm M [14] differs from W primarily by taking as input partial information on the result type of the expression. Lee and Yi proved that M finds errors "earlier" (this measure is based on the number of recursive calls of the algorithm) than W and claimed that the combination of these two algorithms "can generate strictly more informative type-error messages than either of the two algorithms alone can". Other W variants are W$'$ [16] or UAE [29]. McAdam claims that W suffers a left-to-right bias and proposes to eliminate the bias replacing the unification algorithm used in the application case of W by another one called "unification of substitutions". Yang claims that the primary advantage of UAE is that it also eliminates this left-to-right bias. As explained by McAdam the left-to-right bias in W arises because in the case of an application, the substitution computed for the first component of the application is applied to the type environment when type-checking the second component.

All the algorithms mentioned above retain a left-to-right bias in handling of let-bindings and they all blame only one syntax tree node for the failure of type inference when a set of nodes is at fault. Choosing only one node to report as the error site often leads to poor error reports that identify error locations sometimes far away from the actual programming error locations. The situation is made worse because the node targeted for blame depends on internal implementation details, i.e., the order in which tree nodes are visited and which constraints are accumulated and solved at different points of the traversal. The confusion is further worsened because these algorithms generally exhibit in error messages (1) an internal representation of the program subtree at the identified error location which often has been transformed substantially from what the programmer wrote, and (2) details of inferred types which were not written by the programmer and which are anyway erroneous and therefore confusing.

Improved error reporting systems. Attempting to solve the problem, constraint-based type inference algorithms [21, 22, 23]

separate the two following processes: the generation of type constraints for a given term and the unification of the generated constraints. Many works are based on this idea to improve error reporting (a probably incomplete list includes [12, 8, 9, 6, 24, 25, 26]).

Independently from this separation, there exist many ways to improve error reporting. As explained by Yang et al. [31], there exist different approaches toward improving error reporting: error explanation systems [1, 30] and error reporting systems [27]. The type error reporting systems can then be divided further into subgroups. Another approach to type error reporting is the one of Lerner et al. [15] or Hage and Heeren [10] suggesting changes to perform in the untypable code to solve type errors.

Type error slicing. Haack and Wells [9] noted that "*Identifying only one node or subtree of the program as the error location makes it difficult for programmers to understand type errors. To choose the correct place to fix a type error, the programmer must find all of the other program points that participate in the error.*" They proposed locating type errors at *program slices* which include all parts of an untypable piece of code where changes can be made to fix the type error and exclude the parts where changes cannot fix the error.

Haack and Wells gave their method of *type error slicing* for a tiny subset of SML barely larger than the λ-calculus. The method was given in three main steps: assigning constraints to program points, finding minimal unsolvable subsets in the set of all constraints, and computing type error slices. This method is unbiased in the sense that it first generates type constraints before using a unification algorithm on the generated set of constraints. This method also meets all the criteria listed in [31] for error reports to be "good": it reports only errors for ill-typed pieces of code (*correct*), it reports no more than the conflicting portions of code (*precise*), it reports short messages (*succinct*), it does not report internal information such as internal types generated during the type inference process (*amechanical*), it does report only code written by the programmer, and the code does not go through transformations as it is the case when using existing SML implementations (*source-based*), it does not privilege any location over the others (*unbiased*), and it reports all the conflicting portions of code (*comprehensive*).

In this approach, their type error slicer enumerates all the minimal unsatisfiable subsets of the set of constraints generated for a given piece of code. A type constraint is labelled by the location responsible for the generation of the constraint. These labels are then used to identify minimal sets of locations corresponding to type errors. The computation of slices is based on these sets of labels: a slice corresponds to a piece of code where all the sub-terms not involved in a particular error (a minimal set of locations) are replaced by some visually convenient symbols (such as dots). These slices are intended to contain all and only what the programmer needs to solve the type errors. Moreover, Haack and Wells also implemented highlighting of the slices in the original piece of code.

Slicing for a full language. Our goal is a type error slicer that (1) covers the full SML language and (2) is practical on real programs. As would happen for any programming language, while doing this we encountered challenges.

One challenge was to find a suitable constraint system to avoid a combinatorial explosion of the number of constraints during the constraint generation. A naive approach to constraint generation might involve the duplication of the environment of a polymorphic binding such as in a let expression. We decided to base our constraint system on the one defined by Pottier and Rémy [23, 22]. The most interesting constraints in their constraint system are "let" constraints. These constraints are called "let" constraints because they are the ones generated for let bindings. They are to some extent, inspired by the constraint-based type systems such as the one by Odersky, Sulzmann and Wehr [21] (and mainly by the type schemes used in this system). The main reason for adopting these constraints is efficiency. As explained by Pottier these forms of constraint "allow building a constraint of linear size" [22]. Pottier also explains that the main idea is to first simplify the constraints associated to the left-hand-side (between `let` and `in`) of a let binding before performing any context duplication. It appears that the structure can be extended to deal with diverse identifiers (polymorphic as well as monomorphic bindings): values, type names and structure names. Moreover the structure of theses constraints is well suited to handle, for example, the value polymorphism feature of SML.

An interesting issue caused by using "let" constraints was making a fast minimisation algorithm that worked with them. The fact that we have to solve the constraints associated to the left-hand-side of a let-binding before any other constraints interferes with a naive approach for a minimisation algorithm that consists of starting from a type error and building a unifier containing the type information about the locations that have to be in the currently built minimal error. These locations cannot be picked as the last locations treated before the failure of the unification algorithm. We designed a new minimisation algorithm that uses another approach. Roughly speaking, in a piece of code, it usually turns out that not all the declarations are involved in a type error. Thus, our minimisation algorithm then tries to unbind declarations through the code to test if they have to be part of the error currently being minimised.

Another challenge is SML's value identifier statuses. In SML, a value identifier can either be a value variable (the only status considered by Haack and Wells) or a datatype constructor or an exception constructor.[1] For example, under the assumption that unknown free identifiers have value variable status, `fn c => (c 1, c ())` is erroneous and the unique minimal error in this piece of code contains the three occurrences of `c` constrained to have the same type, and the two applications of `c` to the two different arguments `1` and `()`, which are constrained to have different types (`int` and `unit` respectively). However this minimal error does not exist if the code is preceded (for example) by `datatype t = c`. The three occurrences of `c` would not then be constrained to have the same type by the `fn`-binding and a minimal type error would then be, among others, that `c` is declared as a datatype constructor (occurs in a datatype declaration) without argument and that the second occurrence of `c` in our example is applied to an argument.

We solve this challenge by generating constraints annotated by context dependencies. For example in `fn (x, _) => x + 1` we generate a constraint that relates the two occurrences of `x` annotated by the dependency that `x` is a value variable and not a datatype constructor (or exception constructor). This constraint is then discarded if a context is found that confirms that `x` must be a datatype constructor. The dependency is discarded and the constraint is kept if a context is found that confirms that `x` cannot be a datatype constructor (i.e., if `x` is bound by a recursive declaration which, as defined in the Definition of Standard ML [19], can overwrite the status of an identifier). Considering that most of the time the identifiers bound by a function are value variables, we report conditional type errors that assume that unbound identifiers are considered to be value variables and not datatype constructors. We could also report errors assuming that these identifiers are datatype constructors, but we do not because most of the time these identifiers are value variables and we would cause a great increase in unhelpful reported slices.

In addition to overcoming challenges like those above, we also extend the method to cover context-sensitive syntax errors. This extension comes naturally from handling identifier statuses, because for example if a piece of code is studied out of its context then an

[1] In the formal part of this article, we omit treatment of exception constructors because they are handled very similarly to datatype constructors.

identifier can generate a syntax error depending on its status if it occurs twice in a pattern. In `fn (x, x) => x`, the fact that `x` occurs twice in the pattern is an error only if `x` is a value variable but is not if `x` is a datatype constructor.

We also improve the notions of highlighted pieces of code and slices so that, for example, a programmer immediately knows if it is important for a piece of code to be a direct subtree of its context. In the code written by the programmer, our solution is to provide a meaningful highlighting of white spaces.

Other challenges include handling SML features (e.g., value polymorphism or the implicit binding of explicit type variables) in order that type error slices involving these features contain enough information to be understandable.

Moreover, our approach is fully compositional: the slicing operation can be operated on a piece of code containing free variables and context dependencies always have the potential to be discarded.

To guide the design of our constraint generator, we have developed an alternate type system for SML in which polymorphism is not expressed using the "for all" ($\forall$) quantifier as it is usually done in the literature [18, 19, 7]. Instead, the way we deal with polymorphism is inspired by intersection types [3]. Because the type error slicer does not need the type system (it is only used in guiding its design), we do not present it except in our technical report.

2. Type error slices for SML

This section explains our method and illustrates it with examples.

We sometimes compare our reports to the reports of some SML compilers. These compilers are the version 110.52 of SML/NJ, the version 1.3.0 of HaMLet and the version 2.01 of Moscow ML.

Note that the implementation of our slicer deals with many more kinds of errors than the ones formally presented in Section 4. However, this informal section shows some examples involving kinds of errors only handled by our implementation. Also, some kinds of errors were already present in Haack and Wells's type error slicer [9], but these are now augmented with extra information: context dependencies on the status of the free identifiers.

The colour convention we use is that a piece of code is highlighted in red (or very light grey) if it is involved in the reported error; a piece of code is highlighted in purple or blue (or dark grey) if the error reported is a clash (between two type constructors for example) and the code is an end point in the error (one of the clashing type constructors for example); finally a piece of code is highlighted in green (or light grey) if it is a record field that occurs in more than one minimal slice of a same record clash.

2.1 Non context dependent type errors

In this section we present non context dependent type errors; we introduce the use of boxes to highlight parts of code that do not have explicit tokens; and we briefly present our syntax for slices.

Let us start by presenting two simple examples involving datatype definitions leading to **type constructor clashes** (with the error locations reported by SML/NJ enclosed in boxes):

```
fun ex1 z = let datatype X = C1 of int in (C1 true) end
fun ex2 z = let datatype Y = C2 | C3 of int in (C2 z) end
```

For both (untypable) functions presented above SML/NJ reports only one location as the error location. We make these locations explicit in the code by circling them. Note that for example in the first function the programmer can solve the type error by replacing `int` by `bool` and that this location is not reported.

In `ex1`, the datatype constructor `C1` is applied to a Boolean but is defined as containing an integer. We obtain a clash between the integer and Boolean types. The three compilers SML/NJ, HaMLet and Moscow ML report that the type of the argument of `C1` is not equal to the type of its domain.

In `ex2`, the datatype constructor `C2` is applied to an argument but is not defined as taking an argument. As for the first example, SML/NJ, HaMLet or Moscow ML only report `C2` (or `C2 z`) when `C2`'s definition might be the programming error location (the location where the programmer will eventually make some changes).

Our type error slicer exhibits the two following highlightings:

```
fun ex1 z = let datatype X = C1 of int in C1 true end
fun ex2 z = let datatype Y = C2 | C3 of int in C2 z end
```

In this example, we print a box around the last occurrence of `z` because highlighting the space between `C2` and `z` makes sense only because `z` is the argument of `C2`. This is one of our uses of boxes: around arguments (whose content does not matter but whose presence does matter). Note that in our approach the highlighting of the white spaces between a function and its argument is important (the highlighting of the white spaces in general) because these white spaces are one of the two end points of the second highlighting presented above. This application constrains `C2` to have an arrow type. Note that if we had `C2(z)` instead of `C2 z`, the box around the argument `(z)` would be marked as the end point.

We also provide the error reports using slices intended to contain all and only the parts involved in the reported errors. The dots are used to show that some irrelevant information for the errors to occur have been removed. Given a term and a type error in this term, the irrelevant information for this error is precisely the set of nodes and edges in the abstract syntax tree of the term not involved in the error. The angle brackets with more than `..` inside are used to indicate nesting at unfixed depth. The two slices associated to the two errors presented above are as follows:

```
⟨..datatype ⟨..⟨..⟩ = ⟨..C1 of ⟨..⟩ int..⟩..⟩..C1 true..⟩
⟨..datatype ⟨..⟨..⟩ = ⟨..C2..⟩..⟩..C2 ⟨..⟩..⟩
```

Note that many parts of the original terms are sliced out, such as the names of the datatypes which are not involved in the errors. These carefully defined slices are intended to contain all and only the information needed by the programmer to solve its errors. For example in the first slice, `⟨..C1 of ⟨..⟩ int..⟩` stresses that the number of datatype constructors defined along `C1` is not important and that we do not need to know the arity of `int`. It is not important that `C1` is the only type constructor in a datatype definition and it is not important that the arity of `int` is 0 (the type name `int` can be rebound in SML). Note that both errors are context independent.

To illustrate another use of boxes, let us now present an example illustrating an **arity clash** between the two occurrences of `w`:

```
datatype 'a t = T of (bool -> ('a, 'b) w) w
```

In this context, the box is used to stress the importance that `(bool -> (('a, 'b) w))` is a type sequence of length 1 but that the type by itself (what is inside the box) might not be important for the error. These boxes are needed because type errors do not always involve explicit tokens. There are not always explicit tokens expressing the arity of a type name. In the example above we could use the parentheses around `bool -> ('a, 'b) w`, but in `('a, 'b w) w` which generates a similar error, there are no parentheses we could use around `'b`. Moreover, some information might be important inside the type, as it is the case in our example. That is why we obtain highlighted code inside the box.

In `fun f () = f () 0`, we clearly obtain a **circularity** problem when trying to infer a type for `f`. SML/NJ reports a circularity problem and the following explanation of the problem:

```
Error: right-hand-side of clause doesn't agree with
       function result type [circularity]
expression: 'Z
result type: unit -> 'Z
in declaration:
  f = (fn () => (f <exp>) ())
```

In this report `'Z` is an internal type variable of SML/NJ's type checker that the programmer never wrote and the reported code `f = (fn () => (f <exp>) ())` is not what the programmer wrote either. Programmer's code is transformed by SML/NJ's compiler and the reported code is then the transformed code, which might be confusing for the programmer. This arises because SML's `fun` feature is what is called a "derived form" and is equivalent to a form of what is called the "bare language" (see the Definition [19]), but the forms handled by SML/NJ's type checker are eventually only the forms of the bare language, the derived forms being transformed into their equivalent forms during the "elaboration" phase.

2.2 Context dependent type errors

Let us now present errors with context dependencies. The next examples also show some of the details stressed by our slices for recursive functions defined using the `fun` feature. We present here a small piece of code, but we could easily imagine the functions `g` and `f` being far away in a bigger code. Let us consider the two following highlightings for the same piece of code:

```
fun g x y z = if z then x + y else y
fun f []    y = y
  | f [x]   y = g x y y
  | f (x :: xs) y = x + (f xs y)

fun g x y z = if z then x + y else y
fun f []    y = y
  | f [x]   y = g x y y
  | f (x :: xs) y = x + (f xs y)
```

This code is well-typed if the last occurrence of `y` in the third line is replaced by a Boolean. We call this occurrence of `y` the programming error location. This is, for the sake of our example, the location where the programmer will eventually make changes to obtain a well-typed program. It appears that for this example the programming error location is contained in every returned slice. Both errors are due to clashes between type constructors. However, the first slice is due to the fact that `f` returns a Boolean (it returns its second argument and it calls `g` where its second argument is used as a Boolean) and also returns an integer (because it returns a call to `g` which returns an integer). The end points of the second slice are the same as for the first slice, namely: the `if then else` and the `+`. However, in the second slice we do not need to know, for example, that `g` returns an integer (there is no highlighting of the white spaces between `=` and `if`, between `then` and `x` and between `y` and `else`). What we need to know however is that there is somewhere a call to `g` where the types of its second and third arguments are constrained to be equal (being a parameter of a function, `y` is assigned a monomorphic type) and that in `g`'s definition, its second argument has to be an integer (because here `+` has the type it is given by the standard basis) while its third argument has to be a Boolean (because of the conditional). We can observe with this example that from a unique problem, different slices can be derived.

We can also see that all these pieces of information appear in our slices: the part of the first slice corresponding to the body of `g`'s definition is `if z then ⟨..⟩ + ⟨..⟩ else ⟨..⟩` when the same part for the second slice is `⟨..if z then ⟨..⟨..⟩ + y..⟩ else ⟨..⟩..⟩`

Note that the dots do not necessarily replace explicit tokens. We consider extra nodes, each of them corresponding to the link between the node associated to a term and the node associated to one of its direct subterms. In the last partial slice above, we removed the extra node linking the conditional to its `then` branch for the reason explained above. Another interesting example concerns the part of the second slice corresponding to the third line of the code: `⟨..⟩..⟨..y..⟩..= ⟨..g ⟨..⟩ y y..⟩` It means that we do not need to know the name of the function and that it has at least one argument in which appears the identifier `y`. Now, the circled dots on the left of the equal symbol mean that it is irrelevant to know if this function has more arguments. These dots do not match any piece of code because this function has no more arguments after `y`.

Both highlightings are obtained under the context dependencies that `y` and `z` are value variables. As a matter of fact, if we added the datatype declaration `datatype t = y` before this code, we would not obtain these two slices but we would, among others, obtain a slice presenting the clash between the types of the two branches of the conditional in `g`'s definition. If we assume that `+` is a function from a pair of integers to an integer then the `then` branch of the conditional would return an integer while the `else` branch would return a `t`. We would obtain the following slice (among others):

```
⟨..datatype ⟨..⟨..⟩ t = ⟨..y..⟩..⟩
 ..if ⟨..⟩ then ⟨..⟩ + ⟨..⟩ else y..⟩
```

2.3 Syntactic errors

Let us now present some (context-sensitive) syntactic errors.

The interesting point in the following piece of code is that the application of `g` to an argument is an error if `g` is a non-constructor but might not be an error if `g` is defined somewhere as a constructor. We report the error that **an identifier is applied to an argument in a pattern** under the context dependency that `g` is a value variable

```
datatype t = f of int; fn (f, f y, g x) => x + y
```

Note that this error would not be context-sensitive if, for example `fun g x = x + 1` was a declaration preceding the code presented above. This declaration of `g` would then be part of the error.

We obtain a second error reporting that two occurrences of `f` are constrained to have a functional and non-functional type. More generally the error is that **an identifier occurs in a pattern both applied and not applied to an argument**:

```
datatype t = f of int; fn (f, f y, g x) => x + y
```

As we can see in this highlighting, this error would also occur without the datatype declaration because if `f` was a value variable then the second highlighted occurrence of `f` would constrain it to be a datatype constructor. If `f` was a datatype constructor, as explained above it would be used both at locations constraining its type to be functional and non-functional.

2.4 Where our type error slicer becomes absolutely vital

Let us now consider a more complicated example (with the programming error circled and SML/NJ's error location in an oval). Through this example we will see that our type error slicer becomes vital for more complicated errors.

```
datatype ('a, 'b, 'c) t = Red    of 'a * 'b * 'c
                        | Blue   of 'a * 'b * 'c
                        | Pink   of 'a * 'b * 'c
                        | Green  of 'a * 'b * 'b
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red    (x, y, z)) = Blue   (y, x, z)
  | trans (Blue   (x, y, z)) = Pink   (y, x, z)
  | trans (Pink   (x, y, z)) = Green  (y, x, z)
  | trans (Green  (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red    (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x), #2 x)
```

The code declares a datatype to deal with colours and a function manipulating these colours. This function is then used on an instance of a colour (the first element in the pair `x`). The expected type of `y` is `(int, bool) u`. Let us assume that in this piece of code we wrote `'b` instead of `'c` in `Green`'s definition. This location, called the programming error, is circled in the piece of code.

SML/NJ reports a type constructor mismatch error (the portion inside the oval in the piece of code presented above):

```
operator domain: (int,int,int) t
operand: (int,int,bool) t
in expression:
  trans ((fn 1=<pat>,... => 1) x)
```

First, the programming error location is not reported. Then, the reported code does not look like our code and is far away from the programming error location. It is then hard to find where we made an error. As a matter of fact, we would have obtained the same error message if instead of the error described above, we had written `x` instead of `z` in the right-hand-side of any of `trans`'s branch. With SML/NJ's report we would have to check the entire piece of code.

Moscow ML reports also a type constructor clash as follows and blames the same portions as SML (and HaMLet):

```
File "test-prog.sml", line 15, characters 25-37:
! val y : (int, bool) u = (trans (#1 x), #2 x)
!                          ^^^^^^^^^^^^
! Type clash: expression of type
!   (int, int, int) t
! cannot have type
!   (int, int, bool) t
```

The first highlighting obtained with our type error slicer is:

```
datatype ('a, 'b, 'c) t = Red    of 'a * 'b * 'c
                        | Blue   of 'a * 'b * 'c
                        | Pink   of 'a * 'b * 'c
                        | Green  of 'a * 'b * 'b   ①
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red    (x, y, z)) = Blue   (y, x, z)
  | trans (Blue   (x, y, z)) = Pink   (y, x, z)
  | trans (Pink   (x, y, z)) = Green  (y, x, z)   ②③
  | trans (Green  (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red    (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)   ⑤
val y : (int, bool) u = (trans (#1 x), #2 x)
                                       ④
```

This error is context-sensitive: it is obtained under the dependencies that `y` and `z` are value variables. Note that the programming error location is in the slice. We can then search for our error looking only at the highlighted portions of code. We can start with the end points. We note that the type `(int, bool) u` constrains the type of the application of `trans` to an argument and that the highlighted portion of `trans` is when applied to `Green`. At ①, the second argument of `Green` is constrained to be of the same type as its third argument. At ②, `y` is incidentally constrained to be of the same type as `z`. At ③, because `y` and `z` are respectively the first and third arguments of `Yellow` and using the definition of `Yellow`, we can deduce that the type of the application of `Yellow` to its three arguments and so the result type of `trans` is `t` where its first (`'a`) and third (`'c`) parameters have to be equal. Using the code at ④ and ⑤ we can see that the result type of `trans` is constrained to be of type `t` where its first (`int`) and third (`bool`) parameters are different.

We can see that without the highlighting it would be hard to guess why we obtain a conflict between `(int,int,int) t` and `(int,int,bool) t`.

2.5 Merging of minimal slices

It is sometimes convenient to merge different minimal errors. We propose a solution to merge record clashes.

Let us consider the two following reports of **record clashes** in the same piece of code: `val {foo,bar} = {fool=0,bar=1}`

This slice is obtained because `fool` is not in the set $\{\texttt{foo}, \texttt{bar}\}$ and because `foo` is not in the set $\{\texttt{fool}, \texttt{bar}\}$. In this case it is more convenient to present a set of minimal slices with only one highlighting. As explained, light grey is used to highlight the fields that are common to different minimal slices. For merged slices minimality is understood as follows: retain a single dark grey label in one of the two clashing records and all labels in the other.

SML/NJ reports a type constructor mismatch:

```
pattern: {bar:'Z, foo:'Y}
expression: {bar:int, fool:int}
in declaration:
  {bar=bar,foo=foo} =
    (case {fool=0,bar=1}
      of {bar=bar,foo=foo} => (bar,foo))
```

As we can see this error report contains type variables that have not been written by the programmer and the reported code is confusing because it differs from what the programmer wrote.

HaMLet reports only a "type mismatch between pattern and expression" and Moscow ML reports that `foo` is missing in the type of the expression `{fool=0,bar=1}`.

3. Mathematical definitions and notations

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers and let i, j, k, n, m, p, q be metavariables ranging over $\mathbb{N}$.

If a metavariable v ranges over a class C, then the metavariables v_x (where x can be anything) and the metavariables v', v'', etc., also range over C.

Let s range over sets. The power set of a set is defined as usual: $\mathbb{P}(s) = \{s' \mid s' \subseteq s\}$, where $\subseteq$ is the usual set inclusion. If v ranges over s, $\tilde{v}$ is defined to range over $\mathbb{P}(s)$.

A binary relation is a set of pairs. Let R range over binary relations. Let $\mathrm{dom}(R) = \{x \mid (\!|x, y|\!) \in R\}$ and $\mathrm{ran}(R) = \{y \mid (\!|x, y|\!) \in R\}$ (note that the pair $(\!|x, y|\!)$ differs from the tuple (defined below) of length 2, $\langle x, y\rangle$). A function is a binary relation f such that if $\{(\!|x, y|\!), (\!|x, z|\!)\} \subseteq f$ then $y = z$. Let f range over functions. Let $s \to s' = \{f \mid \mathrm{dom}(f) \subseteq s \wedge \mathrm{ran}(f) \subseteq s'\}$. Let $x \mapsto y$ be an alternative notation for $(\!|x, y|\!)$ used when writing functions.

A tuple t is a function such that $\mathrm{dom}(t) \subset \mathbb{N}$ and if $1 \leq k \in \mathrm{dom}(t)$ then $k - 1 \in \mathrm{dom}(t)$. Let t range over tuples. We write the tuple $\{0 \mapsto x_1, \ldots, n-1 \mapsto x_n\}$ as $\langle x_1, \ldots, x_n\rangle$ and use $\langle\rangle$ for the empty tuple, i.e., $\langle\rangle$ is an alternative notation for $\varnothing$ used when writing tuples. We say that $\langle x_1, \ldots, x_n\rangle$ is a tuple of length n. We define the appending $\langle x_1, \ldots, x_n\rangle @ \langle y_1, \ldots, y_m\rangle$, of two tuples $\langle x_1, \ldots, x_n\rangle$ and $\langle y_1, \ldots, y_m\rangle$ as the tuple $\langle x, \ldots, x_n, y_1, \ldots, y_m\rangle$. Given a set s, let $\mathrm{tuple}(s) = \{t \mid \mathrm{ran}(t) \subseteq s\}$. If v ranges over s, $\vec{v}$ is defined to range over $\mathrm{tuple}(s)$. Given n sets $s_1, \ldots, s_n$, let $s_1 \times \cdots \times s_n$ stand for the set of all the tuples $\langle x_1, \ldots, x_n\rangle$ such that $x_i \in s_i$ for all $i \in \{1, \ldots, n\}$.

4. Technicalities

This section presents a restricted version of our type error slicer implementation. The method followed by our slicer consists in three main steps: constraint generation, enumeration and minimisation of label sets representing errors, and slicing. The enumeration and minimisation algorithms both make an extensive use of a unification algorithm.

4.1 Syntax

We present a subset of our syntax in Figure 1. Note that some terms are surrounded by the brackets $\lceil$ and $\rceil$ and that we use ϵ to stand for an empty sequence. These symbols are not part of the syntax considered by the programmer but they are part of an internal

representation of the syntax. The brackets exist only to provide a visually convenient place to put a label and ϵ exists only to provide a visually convenient display of an empty sequence. We use $\lceil\ \rceil$ to avoid confusion with ( ) as part of SML syntax.

All the labels such as l_1 and l_2 in $\lceil {ty_1}^{l_1} \times {ty_2}^{l_2} \rceil^{l}$ are associated to context information. For instance, the label l_1 is associated to the context of the type ty_1, meaning the space between ty_1 and the first $\times$. A singleton type variable sequence is ltv^{l} (i.e., ${tv'}^{l}$) because the label occurring in ltv is associated to the type variable occurring in ltv itself and l is associated to the singleton type variable sequence.

The syntax considered in our implementation contains additional features such as SML's `fun` feature, exceptions, records, and many others. As explained before, the context-sensitive syntactic restrictions [19] on the language (e.g., no pattern may contain the same variable twice) are dealt with using slices. An issue is that in a partial piece of code (which can be part of a bigger piece of code) we might not know if an identifier is a value variable or a datatype constructor. It is then impossible to impose the restriction that no pattern may contain the same variable twice. For example it might happen that the following piece of code: `fn (x, x) => x`, is part of a bigger piece of code where `x` is defined as a datatype constructor. Hence, the fact that `x` occurs twice in the pattern part of the piece of code might not be a syntactic error.

4.2 Type system

As part of designing our type error slicer, we developed an alternate presentation of SML's type system [19] which accepts the same programs and program fragments. The type system uses intersection types to more closely correspond to the bottom-up way we generate constraints. This type system is not needed for doing type error slicing, but mainly serves to relate our constraint generator to SML typability. When the set of constraints generated for a piece of code is solvable then the piece of code is typable by our system which is equivalent to SML typability. (For a complete definition of our type system, see Appendix B.)

4.3 Constraints

Syntactic forms used in our constraints. Figure 2 presents the syntax of type environments (top of Figure 2) and constraints (bottom of Figure 2, where the prefixing I in set names stands for Internal) used by our constraint generator. One important point of our syntax is the introduction of type sequences (ω), corresponding to, for example, `(int,bool)` from the type `(int,bool)t` in SML. These types help catch arity clashes. Note that the ω and μ that are not variables contains labels. These labels are used to distinguish end points in clashes (for more details on how this works, see rule (5) of our unification algorithm presented in Appendix D). Other important syntactic forms are the constraints on identifiers (υ) which are meant to be used in our type environments to constraint the type of an identifier. A constraint on identifier contains two type variables in order to record extra information on identifiers: for a value identifier, we need the second type variable because the type of a value variable and the type of a datatype constructor are different if the identifier occurs inside a pattern. The first type variable constrains the type of the identifier in both cases and the second type variable is complementary information in case the identifier is a datatype constructor. For instance, in a pattern if an identifier is not applied to an argument (and thus might be a value variable or a datatype constructor) and if it turns out that this identifier is a datatype constructor then its type has to be user defined type constructor (it cannot be, for example, an arrow type). We use the second type variable to add this information: a constraint is generated specifying that this second variable is equal to a new type constructor (different from a basic one). For example in `fn c => c`, if $\langle \alpha_1, \alpha_2, l \rangle$ is a constraint on the identifier `c` then α_2 is constrained to be equal to $\langle\langle \delta, l \rangle, \beta \rangle$ where δ is always considered to be different from any basic type constructor so that, for example, it cannot never be equal to the arrow type constructor. If it turns out that `c` is defined as a datatype constructor with an argument (such as in `datatype t = c of int`) then we would obtain a conflict between δ and $\rightarrow$. This second variable is also used to constrain the arity of a type constructor. For example in `('a, 'b) t`, if $\langle \alpha_1, \alpha_2, l \rangle$ is a constraint on the identifier `t` then α_2 is constrained to be equal to $\langle\langle \delta, l \rangle, \langle \alpha, \alpha' \rangle\rangle$ where the length of $\langle \alpha, \alpha' \rangle$ constrains the arity of the type constructor `t` to be two. We cannot use the first type variable for this purpose because the types of two type constructions based upon the same type constructor might be different.

Type environments. An environment (env) for our constraint generator is a tuple containing four environments to which we associate four different meanings. The first environment is used for type names. The second environment is used for datatype constructors (identifiers applied to arguments in patterns). The third environment is used for the bound identifiers that are considered as value variables but might as well turn out to be datatype constructors. The fourth environment is the one for the free identifiers. Note that in any of these environments, each υ in $fenv(vid)$ (resp. $tenv(tc)$) is associated to a distinct occurrence of vid (resp. tc) in the code.

Constraints. We formally present in this paper a subset of the set of type errors handled by our implementation (the ones relevant to our simplified syntax). We present four syntactic errors, each of them is described below and corresponds to one kind of syntactic constraint (CsSyn for syntactic constraints). We present four semantic constraints (CsSem for semantic constraints) used to record the bindings in the studied piece of code ("let" constraints) and to record the constraints between: internal types (LabTy), type sequences (LabSeq) and type names (LabName).

Semantic equality constraints. Our three kinds of semantic equality constraint are annotated by labels corresponding to the regions of the program text responsible for their generation. When initially generated by the constraint generator, such a constraint is annotated by a unique label (the one associated to the precise piece of code responsible for its generation) and no context dependency. Equality constraints generated during the unification process can then be annotated by a set of labels and a set of context dependencies. For example, $\alpha_1 \overset{\{l_1\},\varnothing}{=\!=\!=} \alpha_2$ and $\alpha_2 \overset{\{l_2\},\varnothing}{=\!=\!=} \alpha_3$ can generate during the unification process the following constraint: $\alpha_1 \overset{\{l_1,l_2\},\varnothing}{=\!=\!=} \alpha_3$. Context dependencies can only be generated during the unification process, while dealing with "let" constraints.

Semantics "let" constraints. A "let" constraint is as follows: **let** $\overline{csb}_1, \overline{csb}_2(\overline{c}_1)$ **in** $\overline{c}_2$ **end** where $\overline{csb}_1$ and $\overline{csb}_2$ are the bindings related to the bindings in the studied piece of code. A binding can either be polymorphic (poly) or monomorphic (mono). In a "let" constraint, the first set of bindings is used for the identifiers for which the status is unknown, while the second is used for identifiers with known status (which might be variable or datatype constructor). It happens for the simple language presented here (but not in the implementation of our slicer) that the first set of bindings only contains monomorphic bindings. This is because this paper does not present non recursive value declarations (while the implementation of our type error slicer does consider this feature).

Let us first consider the polymorphic case with the following example: `let val rec f = (fn x => x) in f 1 end`. The first occurrence of `f` binds the second occurrence of `f`. If $\overline{c}_1$ is the set of constraints generated for the `val rec` declaration and $\overline{c}_2$ is the set

Figure 1. Restricted version of the labelled syntax of our implementation

Let us consider lx ranging over LabS containing the terms of the form x^l for any x and S such that $x \in \mathsf{S}$. In this table let $n \geq 1$ and $m \geq 2$.

$$
\begin{array}{rcll}
\iota & \in & \mathsf{Int} & \text{(the set of integers)} \\
l & \in & \mathsf{Label} & \text{(set of labels)} \\
vid & \in & \mathsf{VId} & \text{(set of identifier)}
\end{array}
\qquad
\begin{array}{rcll}
\alpha, tv & \in & \mathsf{TyVar} & \text{(set of type variables)} \\
tc & \in & \mathsf{TyCon} & \text{(set of type constructors)} \\
id & \in & \mathsf{Id} & = \mathsf{VId} \cup \mathsf{TyCon}
\end{array}
$$

$$
\begin{array}{rcll}
tvseq & \in & \mathsf{TyVarSeq} & ::= ltv^l \mid \epsilon^l \mid (ltv_1{}^{l_1}, \ldots, ltv_n{}^{l_n})^l \\
ty & \in & \mathsf{Ty} & ::= tv^l \mid \mathrm{int}^l \mid \mathrm{unit}^l \mid (lty)^l \mid lty_1 \xrightarrow{l} lty_2 \mid \lceil lty_1 \times \cdots \times lty_m \rceil^l \mid \lceil tyseq\ ltc \rceil^l \\
tyseq & \in & \mathsf{TySeq} & ::= ty^l \mid \epsilon^l \mid (lty_1, \ldots, lty_n)^l \\
conbind & \in & \mathsf{ConBind} & ::= vid^l \mid lvid^{l_1}\ \mathrm{of}^{l_2}\ lty \\
conbindseq & \in & \mathsf{ConBindSeq} & ::= conbind_1 \mid \cdots \mid conbind_n \\
datbind & \in & \mathsf{DatBind} & ::= tvseq\ ltc \stackrel{l}{=} conbindseq \\
dec & \in & \mathsf{Dec} & ::= \mathrm{val\ rec}\ lpat \stackrel{l}{=} lexp \mid \mathrm{datatype}\ datbind_1 \ \mathrm{and}\ \cdots\ \mathrm{and}\ datbind_n \\
atexp & \in & \mathsf{AtExp} & ::= vid^l \mid \iota^l \mid ()^l \mid (lexp)^l \mid (lexp_1, \ldots, lexp_m)^l \mid \mathrm{let}^l\, dec\ \mathrm{in}\ lexp\ \mathrm{end} \\
exp & \in & \mathsf{Exp} & ::= atexp \mid \mathrm{fn}^l\ match \mid \lceil exp\ atexp \rceil^l \\
match & \in & \mathsf{Match} & ::= mrule_1 \mid \cdots \mid mrule_n \\
mrule & \in & \mathsf{MRule} & ::= lpat \stackrel{l}{\Rightarrow} lexp \\
atpat & \in & \mathsf{AtPat} & ::= \_ \mid vid^l \mid \iota^l \mid ()^l \mid (lpat)^l \mid (lpat_1, \ldots, lpat_m)^l \\
pat & \in & \mathsf{Pat} & ::= atpat \mid \lceil lvid\ atpat \rceil^l
\end{array}
$$

Figure 2. Environments and constraints

$$
\begin{array}{rcll}
\upsilon & \in & \mathsf{CsId} & ::= \langle \alpha_1, \alpha_2, l \rangle \\
fenv & \in & \mathsf{VIdEnv} & = \mathsf{VId} \to \mathbb{P}(\mathsf{CsId}) \\
tenv & \in & \mathsf{TyConEnv} & = \mathsf{TyCon} \to \mathbb{P}(\mathsf{CsId}) \\
env & \in & \mathsf{Env} & = \mathsf{TyConEnv} \times \mathsf{VIdEnv} \times \mathsf{VIdEnv} \times \mathsf{VIdEnv}
\end{array}
$$

$$
\begin{array}{rcll}
\gamma & \in & \mathsf{ITyName} & \text{(set of type names)} \\
\delta & \in & \mathsf{ITyNameVar} & \text{(set of type name variables)} \\
\beta & \in & \mathsf{ISeqVar} & \text{(set of sequence variables)} \\
itcb & \in & \mathsf{ITyConBase} & = \{\to, \times, \mathrm{int}, \mathrm{unit}\} \\
itc & \in & \mathsf{ITyCon} & = \mathsf{ITyName} \cup \mathsf{ITyConBase} \\
itcv & \in & \mathsf{ITyConVar} & = \mathsf{ITyCon} \cup \mathsf{ITyNameVar}
\end{array}
\qquad
\begin{array}{rcll}
\omega & \in & \mathsf{LabSeq} & ::= \beta \mid \langle \overline{\alpha}, l \rangle \\
\mu & \in & \mathsf{LabName} & ::= \langle itcv, l \rangle \\
\tau & \in & \mathsf{LabTy} & ::= \alpha \mid \langle \mu, \omega \rangle \\
poly & \in & & \{\mathrm{poly}, \mathrm{mono}\} \\
csb & \in & \mathsf{CsBind} & ::= \langle id, \alpha_1, \alpha_2, poly, l \rangle
\end{array}
$$

$$
\begin{array}{rcll}
semc & \in & \mathsf{CsSem} & ::= \tau_1 \overset{\overline{l},\overline{vid}}{=\!=} \tau_2 \mid \mu_1 \overset{\overline{l},\overline{vid}}{=\!=} \mu_2 \mid \omega_1 \overset{\overline{l},\overline{vid}}{=\!=} \omega_2 \mid \textbf{let}\ \overline{csb}_1, \overline{csb}_1(\overline{c}_1)\ \textbf{in}\ \overline{c}_2\ \textbf{end} \\
sync & \in & \mathsf{CsSyn} & ::= \mathrm{mul}(\overline{l}, \overline{vid}) \mid \mathrm{con}(\overline{l}, \overline{vid}) \mid \mathrm{inc}(\overline{l}) \mid \mathrm{app}(\overline{l}) \\
c & \in & \mathsf{Constraint} & ::= semc \mid sync
\end{array}
$$

of constraints generated for `f 1` then a "let" constraint is generated as follows: **let** $\varnothing, \{\langle f, \alpha_1, \alpha_2, \mathrm{poly}, l \rangle\}(\overline{c}_1)$ **in** $\overline{c}_2$ **end** where α_1 is the type variable associated to the second occurrence of `f`, α_2 is the type variable associated to the first occurrence of `f`, poly says that this binding is polymorphic and l is the label associated to the first occurrence of `f`. The unification algorithm tries to build a unifier from a constraints set. When dealing with the constraint presented above, the unification algorithm first tries to unify $\overline{c}_1$. If it succeeds, it goes on creating a new constraint from the specified binding. In order to do so, it builds up a type equal to α_2 using the current unifier. If τ is the built type, then $\alpha_1 \overset{\overline{l},\overline{vid}}{=\!=} \tau'$ is created, where τ' is a fresh instance of τ and $\overline{l}$ and $\overline{vid}$ are the set of labels and context dependencies annotating the constraints necessary to build τ. This new constraint is then used to update the unifier. Note that the bindings for a `val rec` do not generate context dependencies. In the simple language reported in this paper, context dependencies can only be generated by a `fn` expression. (Note that in our implementation context dependencies can also be generated by non recursive value declarations because these declarations do not override identifier status.)

Let us now consider a monomorphic binding. Such a binding will be generated for a piece of code such as: `fn x => x + 1`. The first set of bindings in the generated "let" constraint would then be $\{\langle x, \alpha_1, \alpha_2, \mathrm{mono}, l \rangle\}$, where again α_1 is the type variable associated to the first occurrence of `x`, α_2 is the type variable associated to the second occurrence of `x` and l is the label associated to the first occurrence of `x`. The constraint $\alpha_1 \overset{\{l\},\{x\}}{=\!=} \alpha_2$ is then generated. This constraint is used to update the unifier. The unifier also records at this point that α_2 cannot be generalised.

Syntactic constraints. When generated, a $\mathrm{mul}(\overline{l}, \overline{vid})$ constraint (for example for a value variable occurring twice in a pattern) is as follows: $\overline{l}$ contains the two labels of two occurrences of an identifier and $\overline{vid}$ might contain the identifier itself. In the case of two occurrences of a type constructor or of a datatype constructor, because their status is known, $\overline{vid}$ is empty. In the case of an identifier inside a pattern, because the status of the identifier might be unknown, $\overline{vid}$ is then the singleton consisting of the identifier itself. The generated constraint means that there is a multi-occurrence syntactic error only if the identifier is a value variable. If the identifier turns out to be a datatype constructor, the constraint is discarded. If it turns out to be a value variable (bound by a recursive function) then $\overline{vid}$ becomes the empty set and we add the location of this new occurrence of the identifier in $\overline{l}$ (it now contains three locations of the same identifier). When generated, a $\mathrm{con}(\overline{l}, \overline{vid})$ constraint (for a value variable occurring at a function position in a pattern which thus must be a datatype constructor) is as follows: $\overline{l}$ contains the label of an identifier occurring at a function position in a pattern and the label of the application of this identifier to an argument. $\overline{vid}$ contains the identifier itself. If

the identifier turns out to be a datatype constructor, the constraint is discarded. If it turns out to be a value variable (bound by a recursive function) then $\overline{vid}$ becomes the empty set and we add the location of this new occurrence of the identifier in $\overline{l}$. An $\mathrm{inc}(\overline{l})$ constraint (for a free explicit type variable in a datatype declaration) is as follows: $\overline{l}$ contains the label of a type variable occurring free in the datatype declaration and the labels of the abstracted type variables of this datatype declaration (which do not bind the reported free type variable). Such a constraint is context independent: it cannot be modified or discarded. An $\mathrm{app}(\overline{l})$ constraint (for an identifier which is constrained to be both a value variable and a datatype constructor in a pattern) is as follows: $\overline{l}$ contains three labels which are the label of an occurrence of an identifier not at a function position in a pattern, the label of another occurrence of the same identifier at a function position in a pattern and the label of the application of this last occurrence to an argument. Such a constraint is also context independent.

4.4 Constraint generator

Our constraint generator has judgements of the form $x \Downarrow \langle y, z, \overline{c}\rangle$ where x is a piece of program syntax, y is either env, ren (where ren is a renaming of our different variables) or $\langle env, ren\rangle$ and z is one of 9 different kinds of packages of information. $\overline{c}$ is the set of type constraints accumulated so far. y and z allow building constraints for non-local connections.

Most of the judgements in our constraint generator are similar to the ones for expressions: $exp \Downarrow \langle env, \alpha, \overline{c}\rangle$. It expresses that the triple $\langle env, \alpha, \overline{c}\rangle$ is associated to exp using the relation $\Downarrow$.

At the end of the constraint generation, some extra checks are performed: we check if all the occurrences of a free type constructor have the same arity (note that this check has then already been performed for bound type constructors thanks to the use of $\mathrm{checkTy}$ in the rules for let expressions and sequences of datatype bindings in our constraint generator, Appendix C.1) and we check whether an identifier occurs both at datatype constructor and value variable positions in a pattern. (For a complete definition of our constraint generator see Appendix C.)

4.5 Unification algorithm

Our unification algorithm is an extension of the unification algorithm defined by Haack and Wells [9] to cope with our new constraints on type names and sequences and the "let" constraints. It takes as parameter a set of semantic constraints, builds three different unifiers for our three different kinds of variables, keeps track of the monomorphic bindings and either returns these unifiers if it succeeds or returns one of the following type errors (see Figure 3): a constructor clash, a type constructor arity clash or a circularity error. For the clashing errors we record extra information besides the locations of the pieces of code participating in the errors. For a type constructor clash we record the two types clashing and the locations of the corresponding clashing pieces of code (such as the locations of `1` and `true` in `fn x => (x 1, x true)`). Note that a type name variable can participate in such a clash if it turns out during the unification process that it is equal to one of the basic type constructors such as the arrow type constructor. This is because we do not always need to know the name of a type constructor to raise an error. For example if we introduce a new type constructor with a datatype declaration, then it cannot be the arrow type constructor: `let datatype t = T in T 1 end`. In the previous example, `t` does not participate in the error. For an arity clash we record the two arities clashing and their locations. During its process, the unification algorithm generates constraints which accumulate labels and context dependencies from other constraints.

Let us now present our solution to handle "let" constraints through this example (we only present the main steps of the unification process):

```
fn x => let val rec g = (fn h => x h) in (g 1, g true) end.
```

This piece of code is untypable. The first important constraint to be dealt with during the unification process is the one for the top-level function. This constraint looks like **let** $\{csb_1\}, \varnothing(\varnothing)$ **in** $\overline{c}_1$ **end** where $csb_1 = \langle x, \alpha_1, \alpha_2, \mathrm{mono}, l_1\rangle$, α_1 is the type of the first occurrence of `x` and α_2 is the type of its second occurrence. The unification algorithm first records in the current unifier the equality between α_1 and α_2. This equality is annotated by the context dependency `x` (meaning `x` is not a datatype constructor) and the label l_1 (`x`'s label in the pattern). It also keeps track that α_2 cannot be generalised. This information will be discarded after unification of $\overline{c}_1$. The unification algorithm goes on then unifying $\overline{c}_1$ which is the set of constraints for the let declaration. The next interesting constraint the unification algorithm deals with is the one created for the let expression **let** $\varnothing, \{csb_2, csb_3\}(\overline{c}_2)$ **in** $\overline{c}_3$ **end**, where $csb_2 = \langle g, \alpha_3, \alpha_5, \mathrm{poly}, l_2\rangle$, $csb_3 = \langle g, \alpha_4, \alpha_5, \mathrm{poly}, l_2\rangle$, α_3 is the type of the occurrence of `g` applied to `1`, α_4 is the type of the occurrence of `g` applied to `true` and α_5 is the type of the first occurrence of `g`. First, the unification algorithm tries to unify $\overline{c}_2$ which is the set of constraints associated to the `g`'s declaration. The most important constraint in $\overline{c}_2$ is the one associated to the inner function. The constraint looks like **let** $\{\langle h, \alpha_6, \alpha_7, \mathrm{mono}, l_3\rangle\}, \varnothing(\varnothing)$ **in** $\overline{c}_3$ **end** where α_7 and l_3 are the type variable and label associated to the first occurrence of `h` and α_6 is the type variable associated to the second occurrence of `h`. When dealing with this constraint, the unifier is updated specifying that the type α_6 has to be equal to the type α_7 if `h` is a value variable. The unification algorithm also generates at some point when dealing with $\overline{c}_3$ the constraint that α_1 has to be equal to $\alpha_6 \xrightarrow{l_4} \alpha_8$ (which is $\langle\langle\rightarrow, l_4\rangle, \langle\langle\alpha_6, \alpha_8\rangle, l_4\rangle\rangle$) where α_8 and l_4 are the type and label associated to the application `x h`. When $\overline{c}_2$ has been unified, no error has been found yet. Going back to the bindings related to `g`, before building `g`'s type using the current unifier, the unification algorithm recomputes the type variables that cannot be generalised. This is due to the fact that α_2 cannot be generalised, but the unification algorithm does not know yet that α_6 and α_8 cannot be generalised either because of the equality on α_2 and that it is also the case for α_7 because it is equal to α_6. Hence, before the generalisation of `g`'s type, using the current unifier, the unification algorithm builds the types equal to the type variables which currently cannot be generalised. Using these types it then recomputes the set of type variables that cannot be generalised. This set is discarded after the generalisation of `g`'s type. Thanks to this intermediate step, the unification algorithm knows that `g`'s type cannot be generalised. It finally discovers an error when unifying $\overline{c}_3$. (For a complete definition of our unification algorithm see Appendix D.)

4.6 Minimisation and enumeration algorithms

Once a set of constraints has been generated by our constraint generation, we run our enumeration algorithm. This algorithm makes an extensive use of our minimisation and unification algorithms.

Our enumeration algorithm is based on the one defined by Haack and Wells [9]. We modified their algorithm to cope with context dependencies and to incorporate the minimisation process into the enumeration process. We designed a new minimisation algorithm to cope with our new constraint system.

Theoretically, the enumeration algorithm would be sufficient because it is designed to find all minimal errors in a piece of code. Unfortunately, because of the exponential number of potential minimal type errors, an algorithm is needed to minimise an error before searching for another one. It means that when the enumeration algorithm finds an error it is not going to recursively search for all the minimal errors in this found error but for one minimal error as fast as it can and then continue its process.

Figure 3. Error results returned by enumeration algorithm

err	$\in$	Error	::=	$\langle \overline{l}, \overline{vid}, ek \rangle$
ek	$\in$	ErrorKind	::=	$\text{tyConsClash}(\langle l_1, itcv_1 \rangle, \langle l_2, itcv_2 \rangle) \mid \text{arityClash}(\langle l_1, n_1 \rangle, \langle l_2, n_2 \rangle)$
			$\mid$	$\text{circularity} \mid \text{multiOcc} \mid \text{nonConsApp} \mid \text{inclusion} \mid \text{appNotApp}$

The enumeration and minimisation algorithms designed by Haack and Wells [9] were designed to work consecutively. The enumeration algorithm was allowed to run for an certain amount of time before returning the found errors to the minimisation algorithm which would then minimise them. (The found errors would already be minimal if the time limit was not exceeded.)

There are many advantages in having the enumeration algorithm call the minimisation algorithm as soon as it finds an error such as: all errors have to be minimised at some point, so we might as well minimise them as soon as we find them; using Haack and Wells's method, the enumeration algorithm can enumerate two different errors that will then result in the same minimal error (this cannot happen if the minimisation algorithm is called as soon as an error has been found); the search space is reduced more quickly.

We designed a new minimisation algorithm because the form of our "let" constraints interfered with Haack and Wells's minimisation algorithm. The problem was that for a piece of code such as: `let val (x, y) = (true, 1) in x + 1 end`, with the new "let" constraints, we first unify the constraints associated to the declaration. Fresh instances of the types of the declared identifiers are then created and constrained to be equal to the types of the instances of these identifiers in their scope. If an error is not restricted to the body of only one declaration, it can only be discovered when dealing with a "let" constraint. It becomes then complicated to blame a specific label for the failure of the unification (in Haack and Wells's approach such labels are used to build a minimal error).

Our algorithm tries instead in a first phase to remove complete declarations from a found error, making use of the bindings present in the "let" constraints, and then in a second phase to remove any other label that can be removed.

Using "let" constraints is well suited to handle value polymorphism for example. As a matter of fact, in our implementation, a monomorphic binding inside a "let" constraint can depend on a set of labels responsible for the expansiveness of a corresponding expression in the code. During the unification and depending on the input given by the enumeration algorithm (which, during its process, filters out diverse sets of labels, and the corresponding constraints, when calling the unification algorithm), a monomorphic binding can then be changed into a polymorphic one if some dependencies are filtered out.

We have also improved our enumeration algorithms based on other improvements such as the ones developed by de la Banda, Stuckey and Wazny [5]. One of these improvements is that we keep track of the solvable sets of constraints.

Let us now outline the way our enumeration algorithm works. It first enumerates the syntactic errors and then the semantic errors using the semantic constraints and making an extensive use of the unification and minimisation algorithms. The enumeration of the syntactic errors merely consists in the transformation of the syntactic constraints into errors (as defined in Figure 3). During this first phase we also build an initial set of filters (set of labels used to restrict the set of constraints to try to unify) used to restrict the search space. This initial set of filters is either the set of filters obtained during the enumeration of the syntactic errors or if this set is empty and the piece of code is untypable, it is created by using the set of labels obtained by running the unification algorithm on the whole set of constraints and then minimising the found error. Note that if the piece of code is typable then the enumeration stops at this point. Then the enumeration goes on until the set of filters is empty. The minimality of an error is expressed as follows: $err_1 = \langle \overline{l}_1, \overline{vid}_1, ek_1 \rangle$ is smaller than $err_2 = \langle \overline{l}_2, \overline{vid}_2, ek_2 \rangle$ if ($\overline{l}_1 \subset \overline{l}_2$ and $\overline{vid}_1 \subseteq \overline{vid}_2$) or ($\overline{l}_1 \subseteq \overline{l}_2$ and $\overline{vid}_1 \subset \overline{vid}_2$). (For a complete definition of our minimisation and enumeration algorithm see Appendixes E and F.)

4.7 Slicing algorithm

Syntax. First, we extend the syntax presented in Section 4.1 with "dot" terms which are terms of the syntax presented in Figure 1 in which some nodes have been removed.

For example, if we remove the node associated to the label l_2 (the unit atomic expression) in $\lceil 1^{l_1}\ ()^{l_2} \rceil^{l_3}$ then we obtain $\lceil 1^{l_1}\ \text{dots}(\varnothing) \rceil^{l_3}$. Now, if we remove the node associated to the label l_3 (the application) we obtain $\text{dots}(\langle 1^{l_1}, \text{dots}(\varnothing) \rangle)$.

In our implementation, such a term (slice) is displayed as: $\langle .. 1 .. \langle .. \rangle .. \rangle$. The inner dots term being irrelevant, the slicer would then flatten such a term to obtain: $\langle .. 1 .. \rangle$.

Even though the internal syntax for pieces of code and slices is the same, the printing of a term might differ depending if it is a piece of code or a slice. This is the case for matches. Because the bars present in the syntax of a match and the number of match rules are not useful information in a slice they are not exhibited: $\text{fn}^{l}\ 1 \overset{l_1}{\Rightarrow} 1 \mid 2 \overset{l_2}{\Rightarrow} 2$ is then displayed as `fn` $\langle .. $ `1 => 1 .. 2 => 2` $.. \rangle$. (For the definition of the extension of the syntax presented in Section 4.1 with "dot" terms, see Appendix G.)

Flattening. Our slicing algorithm uses two different flattening functions. Both of them flatten nested dot expressions. For example the application of our basic flattening on $\langle .. 1 .. \langle ..$ `false` $.. \rangle .. \rangle$ would result in $\langle .. 1 ..$ `false` $.. \rangle$.

Our second flattening function, the function $\text{seqFlatten}(\overline{seq})$ where seq is either $mrule$ or $conbind$ or $datbind$, considers two different levels. For example, if the function is applied to a sequence of $mrule$, then the first level contains only match rules and the second level any other kind of terms. The flattening of
`fn` $\langle .. \langle ..$ `f` $.. \rangle$ `=>` $\langle .. \rangle .. \langle .. 1 .. \rangle .. \langle ..$ `false` $.. \rangle .. \rangle$ results in
`fn` $\langle .. \langle ..$ `f` $.. \rangle$ `=>` $\langle .. \rangle .. \langle .. 1 ..$ `false` $.. \rangle .. \rangle$ where $\langle .. 1 .. \rangle$ and $\langle ..$ `false` $.. \rangle$ are "dot" match rules (match rules in which the root node has been removed).

Slicing algorithm. The slicing algorithm takes as parameters an expression and a set of labels and slices out all these labels in the expression. The labels provided to the slicer are intended to be the ones generated by the enumerator. (For a complete definition of the slicing algorithm, see Appendix G.)

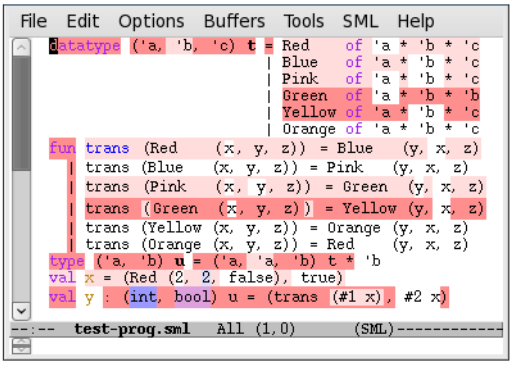
5. Implementation

Language. The implementation of our type error slicer currently handles most of the feature of the core syntax of SML and some of the module features such as basic structures.

Emacs. We currently have a working Emacs interface to highlight slices directly into the source code as it is being edited. We provide a screenshot of the type error presented in Section 2.4 in Figure 4.

The Standard ML basis library. We saw in Section 2 that we use operators such as `::` or `+`. For now, we have two ways

Figure 4. Highlighting of an SML type error in Emacs



of dealing with the Standard ML basis. The first one is that we provide our own predefined basis for these operators. The second one is that we have implemented a way to use library types extracted from a running instance of SML/NJ but there are still technical challenges to overcome. Some types and structures appearing to be hidden in such running instances, one challenge is to get information about type equality. For example, we discover that `+` is overloaded and one of its implementations is `I32.+` at type `?.int32 * ?.int32 -> ?.int32`, but unfortunately there is no structure named `I32` and the type that is supposedly named `?.int32` is probably in fact `Int32.int`. More work is needed, including probably some negotiation with compiler implementers on a standard way of getting this information.

Enumeration. The enumeration uses, during its process, filters on the set of generated constraints. These filters are sets of labels. For example if the enumerator already found an error associated to the two labels l_1 and l_2 then we do not want the enumerator to search again for an error using both the set of constraints labelled by l_1 and the set of constraints labelled by l_2. Thus, the enumerator builds new filters based on l_1 (resp. l_2) to search errors that are not, amongst other things, labelled by l_1 (resp. l_2). The efficiency of the enumerator comes partially from the small size of the set of filters yet to try. However, because of the strategy used by the enumerator it might happen that during its process it tries twice the same filter. Although this can be overcome, it appeared to be at the expenses of the efficiency of the enumerator. However, we improved the efficiency of the enumerator by checking before using a filter $\overline{l}$ that we do not already have found an error that can be generated using $\overline{l}$ instead of just running again the unification using $\overline{l}$ (this checking being faster than running the unification).

6. Related work

In this section we report on similar approaches and briefly compare some of them to our type error slicer.

6.1 Methods making use of slices

After the first version of type error slicing presented by Haack and Wells, many researchers began to present type errors as program slices obtained from unsolvable sets of constraints.

The approach followed by Neubauer and Thiemann [20] uses flow analysis to compute type dependencies for a small ML-like language to then produce type error reports. Based on discriminative sum types, their system can type any term. Type errors can then be produced from the analysis of type derivations. Their method is based on two main steps. The first step called the "collecting phase" consists in the labelling of the studied expression and inferring type information. This analysis generates a set of sets of program points. These program points are directly stored in the discriminative sum types. A conflicting type ("multivocal") is then paired with the locations responsible for its generation. The second step called the "reporting phase" consists in generating error reports from the conflicts generated during the first phase. Slices are built from which highlighting are produced. An interesting aspect of this approach is that a type derivation can then be considered as the description of all type errors in an untypable piece of code, from which another step is needed to compute error reports.

A similar approach to ours is the one by Stuckey, Sulzmann and Wazny [26, 28], which is based on earlier work without slices [24, 25]. They do type inference, type checking and report type errors for the Chameleon language (derived from a Haskell subset). This language includes algebraic data types, type-class overloading, and functional dependencies. As in many recent works, they first code the typing problem into a constraint problem. Their method makes use of labels attached to constraints to keep track of program locations. These labels are then used to highlight parts of untypable pieces of code. First they compute a minimal unsatisfiable set of generated constraints from which they select one of the type error locations to provide their type explanation. They finally provide a highlighting and an error message depending on the selected location. They provide slice highlighting but using a different strategy from ours. They focus more on explaining conflicts in the inferred types as one program point inside the set of error locations. It is not completely clear, but they do not seem to worry much about whether the program text they are highlighting is exactly (no more and no less) a complete explanation of the type error. For example, they do not highlight applications because "they have no explicit tokens in the source code". It is then stated: "We leave it to the user to understand when we highlight a function position we may also refer to its application". This deliberate choice of theirs differs from our strategy in the sense that we think it is preferable to highlight all the responsible parts of a program text responsible for an error even if these parts are white spaces. As another example, they do not appear to highlight the parts of datatype declarations that are relevant to a type error. When running on a translation of the code presented in Section 2.4 into Haskell, ChameleonGecko outputs an error report containing this (the rest of output seems to be internal information computed during the unification process only):

```
ERROR: Type error; conflicting sites:
y = (trans x1, x2)
```

Significantly, because they handle a Haskell-like language, they face a set of technical challenges for accurate type error location that is different from the difficulties with SML.

Gast [6] generates "detailed explanations of ML type errors in terms of data flows". His method is in three steps: generation of subtyping constraints annotated by reasons for their generation; gathering of reasons while unifying the generated constraints; transformation of the gathered reasons into explanations by data flows. He provides, amongst other things, a visually convenient display of the data flows with arrows in XEmacs. Gast's method (which seems to be designed only for a small portion of OCaml) can be considered as a slicing method with data flow explanations.

Braßel [2] presents a generic approach (implemented for the language Curry) for type error reporting based on a method that consists in two different procedures. The first one consists in trying to replace portions of code by dummy terms that can be assigned any type. If an untypable piece of code becomes typable when one of its subtrees has been replaced by a dummy term then the process goes on to apply the same strategy inside the subtree. The second procedure consists in the use of a heuristic to guide the search of

type errors. The heuristic is based on two principles: it will always "prefer an inner correction point to an outer one" and will always "prefer the point which is located in a function farther away in the call graph from the function which was reported by the type checker as the error location". Braßel's method does not seem to compute proper slices but instead singles out different locations that might be the cause of a type error inside a piece of code.

6.2 Significant non-slicing type explanation methods

Milner [17] proved the soundness of the semantics of a small language (application, abstraction, conditional, recursion, local declaration) w.r.t. a typing relation. This result allows to prove that the well typed property is enough to prove the well-defined behaviour of pieces of code. An interesting aspect in Milner's paper is that when giving an informal presentation of his type inference algorithm (W) he separates constraint generation and constraint solving (these two processes are interleaving in the W algorithm which leads to the well-known left-to-right bias).

Heeren et al. designed a method used in the Helium project [12, 11, 13, 10] to provide error messages for the Haskell language relying on a constraint-based type inference. First, a constraint graph is generated from a piece of code. If the piece of code is ill-typed then a conflicting path called an inconsistency is extracted from the constraint graph. Such a conflicting path is a structured unsolvable set of type constraints. Then, they use heuristics to remove an inconsistency. To each type constraint is associated a trust value and depending on these values and the defined heuristics, some constraints are discarded until the inconsistency is removed. They also propose some "program correcting heuristics" used to search for a typable piece of code from an untypable one. Such a heuristic is for example the permutation of parameters which is a common mistake in programming. Their approach has been used with students learning functional programming. Using pieces of code written by students and their expertise of the language they refined their heuristics. They also designed a system of "directives" which are commands specified by the programmer to constrain the set of types derivable from a type class. This approach differs from ours by privileging locations over others by the use of some heuristics. They do not compute minimal slices and highlightings.

We present below the most interesting part of the error report obtained using Helium on a translation of the code presented in Section 2.4 into Haskell. It comes with some warnings (these warnings are not displayed here) on the bindings of identifiers such as the binding of `y` in `trans` (some of these warnings explain, for example, that `y`'s declaration at the end of the code does not bind any of the `y`'s in `trans`'s definition).

```
(16,6): Type error in application
expression       : trans x1
term             : trans
  type           : T a   a   a    -> T a   a   a
  does not match : T Int Int Bool -> T Int Int Bool

Compilation failed with 1 error
```

This error message reports that somehow `x1` and `trans` don't have the expected types. The application is then blamed. Recall that our programming error is at the very beginning of the code and that the Helium's report blames some code at its end.

Lerner, Flower, Grossman and Chambers [15] present a new approach for type error messages. Given an ill-typed piece of code, their method makes use of an unaltered type checker as "an oracle for a search procedure that finds similar programs that do typecheck". The considered language is Caml, but they also developed a prototype for C++. As for Heeren et al. [10], their method consists in the construction of a well-typed program close to a given ill-typed piece of code. The new typable generated code is presented as a possibility of code that the programmer intended to write. Even if the process cannot always find the code the programmer had in mind, it appears that the error locations reported are better than the ones obtained using the usual type checker alone. Other advantages are that the original type checker stays unchanged and does not have to deal anymore with error reporting. Different techniques are used to generate a well-typed program from an ill-typed piece of code. These techniques are: removal of parts of the given code (if such an incomplete code type checks then the removed part is likely to be involved in a type error), constructive change of code (such as switching two parameters, association in different orders, etc.), adaptation to context (the type constraint linking a piece of code to its context is removed) and triage (that tries to "ignore some other parts of the program to focus on one problem" in the case of the presence of more than one error). The focus of their method is not to precisely report on existing type errors but to provide hints that could help the user to turn an untypable piece of code into a typable one. It is a completely different approach from the slicing approach.

7. Summary of contributions

Our contributions in this paper are as follows:

1. We solve a previous efficiency problem (combinatorial explosion of the number of generated constraints) by adopting a new system of constraints in the style of Pottier and Rémy [23, 22]. This constraint system is based on constraints shaped w.r.t. to the studied language: "let" constraints are used to especially deal with let bindings. We design a minimisation algorithm that can handle the new system of constraints.

2. We present details of how to extend type error slicing to deal with problematic aspects of SML. A significant example is dealing with the ambiguity of two distinct identifier statuses in SML (value variable and datatype constructor) while also computing minimal type error slices. We solve this ambiguity by generating type constraints with context dependencies on the status of identifiers. Context dependencies state that some specific identifiers have to be value variables. Slices and highlightings (our two kinds of type error reports) are then provided depending on the context dependencies of the associated unsolvable sets of type constraints. Every dependency has the potential to at any time either be confirmed (and forgotten) or rejected (all constraints depending on this dependency are then discarded).

3. We extend the type error slicing machinery to handle context-sensitive syntactic errors. These include errors that are related to the ambiguity of identifier status, as well as other context-sensitive syntax errors. As for the semantic errors, some of the syntactic errors are generated under context dependencies.

4. We present multiple examples to illustrate our carefully defined unambiguous slices, designed to provide as close as possible to exactly the information needed by the programmer to solve its errors. Our slices and highlightings provide specific details concerning each feature of SML implemented in our type error slicer.

5. We report some technical details on the mechanisms of our type error slicer and discuss different implementation issues encountered during the creation of our slicer.

6. We have an implementation that handles more features than reported in the formalism in this paper. We report in the paper's formalism features such as datatype declarations. Our implementation handles in addition records, tuples, exceptions, type definitions, many more cases of declarations, mutually recursive functions, `fun` syntax, explicit types, value polymorphism, scope of explicit type variables and some of the module features. Our implementation also handles syntactic sugar for lists, conditionals, while loops, case expressions, and sequencing of expressions.

7. A web demonstration of our type error slicer is available at `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicing/slicing.cgi`, and we will soon release its sources.

8. Future work

We have already started to implement the merging of minimal slices. We would like to extend this idea for other kind of errors than record clashes. Merging errors can sometimes be useful to emphasis the locations shared by different minimal slices for the same programming error. This would involve for example the use of different shades of colours depending on the number of minimal slices in which a location contributes. Note that if a location is shared by many slices it does not necessarily mean that it is more likely to be a programming error location.

We are planning on adding extra annotations on constraints which would then generate extra error messages provided to the user along with the highlightings and slices. Such annotations would, for example, explain that a constraint is generated because of a monomorphic binding or because of the value polymorphism restriction or would be some context dependent explanation about the implicit binding of explicit type variable. The produced messages would then give some insight when non obvious features are used and might be responsible for a type error.

In the near future, we plan to finish extending our slicer to a language as close as possible to the full SML language. This includes finishing our implementation of SML's structures and functors (i.e., SML's module system). It also includes features such as flexible records or equality types.

When our type error slicer reaches practical usability on normal programs (i.e., no severe restrictions on the use of SML features), we want to run experiments comparing the effectiveness in improving the productivity of real users of type error slicing vs. more traditional type error messages.

Finally, we have begun user evaluations that will help us design proper experiments.

References

[1] M. Beaven, R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4), 1993.

[2] B. Braßel. TypeHope - there is hope for your type errors. In C. Grelck, F. Huch, G. Michaelson, P. W. Trinder, eds., *Implementation and Application of Functional Languages, 16th Int'l Workshop, IFL 2004*, vol. 3474 of *LNCS*. Springer, 2005.

[3] M. Coppo, M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archive for Mathematical Logic*, 19(1), 1978.

[4] L. Damas, R. Milner. Principal type-schemes for functional programs. New York, NY, USA, 1982. ACM.

[5] M. G. de la Banda, P. J. Stuckey, J. Wazny. Finding all minimal unsatisfiable subsets. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, New York, NY, USA, 2003. ACM.

[6] H. Gast. Explaining ML type errors by data flows. In C. Grelck, F. Huch, G. Michaelson, P. W. Trinder, eds., *Implementation and Application of Functional Languages, 16th Int'l Workshop, IFL 2004*, vol. 3474 of *LNCS*. Springer, 2005.

[7] J.-Y. Girard, P. Taylor, Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[8] C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. In P. Degano, ed., *ESOP*, vol. 2618 of *LNCS*. Springer, 2003.

[9] C. Haack, J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3), 2004.

[10] J. Hage, B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, A. Butterfield, eds., *Implementation and Application of Functional Languages, 18th Int'l Symp., IFL 2006*, vol. 4449 of *LNCS*. Springer, 2007.

[11] B. Heeren, J. Hage. Type class directives. In M. V. Hermenegildo, D. Cabeza, eds., *Practical Aspects of Declarative Languages, 7th Int'l Symp., PADL 2005*, vol. 3350 of *LNCS*. Springer, 2005.

[12] B. Heeren, J. Jeuring, D. Swierstra, P. A. Alcocer. Improving type-error messages in functional languages. Technical report, Utrecht University, 2002.

[13] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005.

[14] O. Lee, K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4), 1998.

[15] B. S. Lerner, M. Flower, D. Grossman, C. Chambers. Searching for type-error messages. In J. Ferrante, K. S. McKinley, eds., *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*. ACM, 2007.

[16] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, A. J. T. Davie, C. Clack, eds., *Implementation of Functional Languages, 10th Int'l Workshop, IFL'98*, vol. 1595 of *LNCS*. Springer, 1999.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[18] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[19] R. Milner, M. Tofte, R. Harper, D. Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[20] M. Neubauer, P. Thiemann. Discriminative sum types locate the source of type errors. In C. Runciman, O. Shivers, eds., *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*. ACM, 2003.

[21] M. Odersky, M. Sulzmann, M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1), 1999.

[22] F. Pottier. A modern eye on ML type inference: old techniques and recent developments. Lecture notes for the APPSEM Summer School, 2005.

[23] F. Pottier, D. Rémy. The essence of ML type inference. In B. C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, chapter 10. MIT Press, 2005.

[24] P. J. Stuckey, M. Sulzmann, J. Wazny. Interactive type debugging in haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2003. ACM.

[25] P. J. Stuckey, M. Sulzmann, J. Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, New York, NY, USA, 2004. ACM.

[26] P. J. Stuckey, M. Sulzmann, J. Wazny. Type processing by constraint reasoning. In N. Kobayashi, ed., *Programming Languages and Systems, 4th Asian Symp., APLAS 2006*, vol. 4279 of *LNCS*. Springer, 2006.

[27] M. Wand. Finding the source of type errors. In *POPL'86: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 1986. ACM.

[28] J. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, University of Melbourne, Australia, 2006.

[29] J. Yang. Explaining type errors by finding the source of a type conflict. In *SFP'99: Selected papers from the 1st Scottish Functional Programming Workshop*, Exeter, UK, UK, 2000. Intellect Books.

[30] J. Yang, G. Michaelson, P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45, 2002.

[31] J. Yang, J. Wells, P. Trinder, G. Michaelson. Improved type error reporting. In M. Mohnen, P. W. M. Koopman, eds., *Implementation of Functional Languages, 12th Int'l Workshop, IFL 2000*, vol. 2011 of *LNCS*. Springer, 2001.

A. More mathematical definitions and notations

This section introduces some mathematical definitions and notations that have not been already introduced in Section 3.

Let $|s|$ be the cardinality of a set s.

Let $\mathrm{dj}(s_1, \ldots, s_n)$ ("disjoint") hold iff for all $i, j \in \{1, \ldots, n\}$, if $i \neq j$ then $s_i \cap s_j = \varnothing$.

Let us now define some functions on relations, functions, and sets:

$$
\begin{array}{lcl}
s \lhd R & = & \{(\!|x, y|\!) \in R \mid x \in s\} \\
s \lhd\!\!\!\cdot R & = & \{(\!|x, y|\!) \in R \mid x \notin s\} \\
f_1 + f_2 & = & f_2 \cup (\mathrm{dom}(f_2) \lhd\!\!\!\cdot f_1) \\
f_1 \uplus f_2 & = & \begin{cases} f_1 \cup f_2 & \text{if } \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) = \varnothing \\ \text{undefined} & \text{otherwise} \end{cases} \\
f_1 \Cup f_2 & = & \begin{cases} (\mathrm{dom}(f_2) \lhd\!\!\!\cdot f_1) \cup (\mathrm{dom}(f_1) \lhd\!\!\!\cdot f_2) \cup (\{x \mapsto f_1(x) \cup f_2(x) \mid x \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2)\}) \\ \qquad \text{if } y \text{ is a set for each } y \in \mathrm{ran}(f_1) \cup \mathrm{ran}(f_2) \\ \text{undefined} \quad \text{otherwise} \end{cases} \\
f^{+x} & = & \{x \mapsto \varnothing\} + f
\end{array}
$$

B. Syntax, types, type environments, constraints and type system

Figure 5 provides the same information given in Figure 1, but with more details. We present the syntax of type environments used by our type system in Figure 6 and we define our type system in Figure 7. Let var range over $\mathsf{Var} = \mathsf{TyVar} \cup \mathsf{ITyName} \cup \mathsf{ITyNameVar} \cup \mathsf{ISeqVar}$.

Functions defined over any of our syntactic forms. Let $\mathrm{someSyntacticForms}(x)$ be the set of syntactic forms belonging to $\mathsf{LabTyVar} \cup \mathsf{Var} \cup \mathsf{Label}$ and occurring in x whatever x is. We then define the following functions:

$$
\begin{array}{lcll}
\mathrm{tyVarSet}(x) & = & \mathrm{someSyntacticForms}(x) \cap \mathsf{TyVar} & \text{(set of syntactic type variables)} \\
\mathrm{lTyVarSet}(x) & = & \mathrm{someSyntacticForms}(x) \cap \mathsf{LabTyVar} & \text{(set of labelled type variables)} \\
\mathrm{tyNameSet}(x) & = & \mathrm{someSyntacticForms}(x) \cap \mathsf{ITyName} & \text{(set of type names)} \\
\mathrm{varSet}(x) & = & \mathrm{someSyntacticForms}(x) \cap \mathsf{Var} & \text{(set of variables)} \\
\mathrm{labSet}(x) & = & \mathrm{someSyntacticForms}(x) \cap \mathsf{Label} & \text{(set of labels)}
\end{array}
$$

Let $\mathrm{djv}(x_1, \ldots, x_n)$ be $\mathrm{dj}(\mathrm{varSet}(x_1), \ldots, \mathrm{varSet}(x_n))$. This function is mostly used by different rules of our constraint generator.

Substitutions over internal types. Note that substitutions are only applied to internal types (σ) which are only used by our type system (and not in constraint solving).

A substitution is defined as follows: $sub \in \mathsf{Substitution} = \mathsf{TyVar} \to \mathsf{InternalTy}$. We apply substitutions on internal types (σ) as follows:

$$
\begin{array}{lcl}
\langle itc, \langle \sigma_1, \ldots, \sigma_n \rangle \rangle[sub] & = & \langle itc, \langle \sigma_1[sub], \ldots, \sigma_n[sub] \rangle \rangle \\
\alpha[sub] & = & \begin{cases} sub(\alpha) & \text{if } \alpha \in \mathrm{dom}(sub) \\ \alpha & \text{otherwise} \end{cases}
\end{array}
$$

We say that σ' is a variant of σ iff there exists sub such that $\sigma' = \sigma[sub]$. We write it $\sigma' \succeq \sigma$.

Renamings of labelled types and equality constraints. Note that renamings are only applied to labelled internal types (τ) and equality constraints (semantic constraints other than the "let" constraints) which are only used for constraint related algorithms (constraint solver, and unification, minimisation and enumeration algorithms). Syntactic constraints and "let" constraints are never renamed.

Let $x_1 \overset{l, \overline{vid}}{=\!=} x_2$ be any of our equality constraints, i.e., where x_1 and x_2 are both either labelled types (τ) or labelled sequences (ω) or labelled names (μ).

A renaming is defined as follows:

$$
\begin{array}{lcl}
ren \in \mathsf{Renaming} & = & \{ren \mid ren = f_1 \cup f_2 \cup f_3 \\
& & \quad \wedge\ f_1 \in \mathsf{TyVar} \to \mathsf{TyVar} \\
& & \quad \wedge\ f_2 \in \mathsf{ITyNameVar} \to \mathsf{ITyNameVar} \\
& & \quad \wedge\ f_3 \in \mathsf{ISeqVar} \to \mathsf{ISeqVar} \\
& & \quad \wedge\ ren \text{ is injective} \\
& & \quad \wedge\ \mathrm{dj}(\mathrm{dom}(ren), \mathrm{ran}(ren))\}
\end{array}
$$

We apply renamings on entities as follows:

$$
\begin{array}{lcl}
\{c_1, \ldots, c_n\}[[ren]] & = & \{c_1[[ren]], \ldots, c_n[[ren]]\} \\
(x_1 \overset{l, \overline{vid}}{=\!=} x_2)[[ren]] & = & x_1[[ren]] \overset{l, \overline{vid}}{=\!=} x_2[[ren]] \\
\langle \langle \alpha_1, \ldots, \alpha_n \rangle, l \rangle[[ren]] & = & \langle \langle \alpha_1[[ren]], \ldots, \alpha_n[[ren]] \rangle, l \rangle \\
\langle itcv, l \rangle[[ren]] & = & \langle itcv[[ren]], l \rangle \\
\langle \mu, \omega \rangle[[ren]] & = & \langle \mu[[ren]], \omega[[ren]] \rangle \\
itc[[ren]] & = & itc \\
var[[ren]] & = & \begin{cases} ren(var) & \text{if } var \in \mathrm{dom}(ren) \\ var & \text{otherwise} \end{cases}
\end{array}
$$

We now define a relation that generates a fresh instance (τ_1 which is regarded as an output) of a type (τ_2):

$$
\mathrm{fresh}(\tau_1, \tau_2, \overline{var}, \overline{var'}) \iff \exists ren.\ \mathrm{dom}(ren) = \mathrm{varSet}(\tau_2) \setminus \overline{var} \wedge \tau_1 = \tau_2[[ren]] \wedge \mathrm{dj}(\mathrm{ran}(ren), \overline{var} \cup \overline{var'})
$$

Figure 5. Restricted version of the labelled syntax of our implementation

(The following syntax recalls the syntax presented in Figure 1 (Section 4.1) using a more readable layout. This figure differs from Figure 1 by the introduction of grammatical rules for $valbind$, $datname$, $datbindseq$, ltv, ltc, $lvid$, lty, $lexp$ and $lpat$. The three first entities were merged inside the definition of a dec in Figure 1. The six other ones formally introduce the metavariables and sets informally introduced by the first sentence in Figure 1.)

ι	$\in$	Int		(the set of integers)	
l	$\in$	Label		(a countable infinite set of labels)	
vid	$\in$	VId		(a countable infinite set of identifier)	
α, tv	$\in$	TyVar		(a countable infinite set of type variables)	
tc	$\in$	TyCon		(a countable infinite set of syntactic type names)	
id	$\in$	Id	$=$	VId $\cup$ TyCon	
ltv	$\in$	LabTyVar	$::=$	α^l	
ltc	$\in$	LabTyCon	$::=$	tc^l	
$lvid$	$\in$	LabId	$::=$	vid^l	
$tvseq$	$\in$	TyVarSeq	$::=$	ltv^l	
			$\mid$	ϵ^l	
			$\mid$	$(ltv_1{}^{l_1}, \ldots, ltv_n{}^{l_n})^l$	where $n \geq 1$
ty	$\in$	Ty	$::=$	tv^l	
			$\mid$	int^l	
			$\mid$	unit^l	
			$\mid$	$lty_1 \xrightarrow{l} lty_2$	
			$\mid$	$\lceil lty_1 \times \cdots \times lty_n \rceil^l$	where $n \geq 2$
			$\mid$	$\lceil tyseq\ ltc \rceil^l$	
			$\mid$	$(lty)^l$	
lty	$\in$	LabTy	$::=$	ty^l	
$tyseq$	$\in$	TySeq	$::=$	ty^l	
			$\mid$	ϵ^l	
			$\mid$	$(lty_1, \ldots, lty_n)^l$	where $n \geq 1$
$conbind$	$\in$	ConBind	$::=$	vid^l	
			$\mid$	$lvid^{l_1}$ of^{l_2} lty	
$conbindseq$	$\in$	ConBindSeq	$::=$	$conbind_1 \mid \cdots \mid conbind_n$	where $n \geq 1$
$valbind$	$\in$	ValBind	$::=$	rec $lpat \stackrel{l}{=} lexp$	
$datname$	$\in$	DatName	$::=$	$tvseq\ ltc$	
$datbind$	$\in$	DatBind	$::=$	$datname \stackrel{l}{=} conbindseq$	
$datbindseq$	$\in$	DatBindSeq	$::=$	$datbind_1$ and $\cdots$ and $datbind_n$	where $n \geq 1$
dec	$\in$	Dec	$::=$	val $valbind$	
			$\mid$	datatype $datbindseq$	
$atexp$	$\in$	AtExp	$::=$	vid^l	
			$\mid$	ι^l	
			$\mid$	$()^l$	
			$\mid$	$(lexp_1, \ldots, lexp_n)^l$	where $n \geq 2$
			$\mid$	let^l dec in $lexp$ end	
			$\mid$	$(lexp)^l$	
exp	$\in$	Exp	$::=$	$atexp$	
			$\mid$	fn^l $match$	
			$\mid$	$\lceil exp\ atexp \rceil^l$	
$lexp$	$\in$	LabExp	$::=$	exp^l	
$match$	$\in$	Match	$::=$	$mrule_1 \mid \cdots \mid mrule_n$	where $n \geq 1$
$mrule$	$\in$	MRule	$::=$	$lpat \stackrel{l}{\Rightarrow} lexp$	
$atpat$	$\in$	AtPat	$::=$	_	
			$\mid$	vid^l	
			$\mid$	ι^l	
			$\mid$	$()^l$	
			$\mid$	$(lpat_1, \ldots, lpat_n)^l$	where $n \geq 2$
			$\mid$	$(lpat)^l$	
pat	$\in$	Pat	$::=$	$atpat$	
			$\mid$	$\lceil lvid\ atpat \rceil^l$	
$lpat$	$\in$	LabPat	$::=$	pat^l	

Figure 6. Internal syntax of types used in the type system (but not in constraints)

st	$\in$	Status	$=$	$\{\mathrm{v}, \mathrm{c}\}$
σ	$\in$	InternalTy	$::=$	$\alpha \mid \langle itc, \overline{\sigma}\rangle$
Γ	$\in$	InternalTyEnv	$=$	$\mathsf{VId} \to (\mathbb{P}(\mathsf{InternalTy}) \times \mathsf{Status})$
∇	$\in$	TypeNameEnv	$=$	$\mathsf{TyCon} \to \mathsf{InternalTy}$
tse	$\in$	TypSysEnv	$=$	$\mathsf{TypeNameEnv} \times \mathsf{InternalTyEnv}$

It takes two additional parameters. The first one ($\overline{var}$) is the set of type variables that cannot be refreshed because they occur in the monomorphic types of the identifiers which contain in their scope the term for which the type is refreshed. The second additional parameter ($\overline{var'}$) is the set of variables that are already used when fresh is called. The type τ_1 is then a fresh instance of τ_2 regarding the computation done prior to the call to fresh. This is used by the function link (note that renamings are also applied in the datatype binding rule of our constraint generator).

Functions on terms of our labelled syntax. The function $\mathrm{seqOfSeq}$ extracts the type variables from a $tvseq$. This function is used by the datatype binding rule of our type system and is defined as follows:

$$\begin{array}{lcl}
\mathrm{seqOfSeq}(tv^l) & = & (\!|tv|\!) \\
\mathrm{seqOfSeq}(ltv^l) & = & \mathrm{seqOfSeq}(ltv) \\
\mathrm{seqOfSeq}(\epsilon^l) & = & (\!|\,|\!) \\
\mathrm{seqOfSeq}((ltv_1{}^{l_1}, \ldots, ltv_n{}^{l_n})^l) & = & \mathrm{seqOfSeq}(ltv_1)@\cdots@\mathrm{seqOfSeq}(ltv_n)
\end{array}$$

Let $\mathrm{flatTyVar}(tv^l)$ be $\{tv \mapsto \{l\}\}$ (this function is used by $\mathrm{mulConsT}$, defined in Appendix C).

Syntactic sugar for types. Let us define some syntactic sugar to deal with types (where $n \geq 2$):

$$\begin{array}{lcl@{\qquad}lcl}
\mathrm{intty} & = & \langle \mathrm{int}, \langle\rangle\rangle & \mathrm{intty}^l & = & \langle\langle \mathrm{int}, l\rangle, \langle\langle\rangle, l\rangle\rangle \\
\mathrm{unitty} & = & \langle \mathrm{unit}, \langle\rangle\rangle & \mathrm{unitty}^l & = & \langle\langle \mathrm{unit}, l\rangle, \langle\langle\rangle, l\rangle\rangle \\
\sigma_1 \to \sigma_2 & = & \langle \to, \langle \sigma_1, \sigma_2\rangle\rangle & \alpha_1 \xrightarrow{l} \alpha_2 & = & \langle\langle \to, l\rangle, \langle\langle \alpha_1, \alpha_2\rangle, l\rangle\rangle \\
\sigma_1 \times \cdots \times \sigma_n & = & \langle \times, \langle \sigma_1, \ldots, \sigma_n\rangle\rangle & \lceil \alpha_1 \times \cdots \times \alpha_n \rceil^l & = & \langle\langle \times, l\rangle, \langle\langle \alpha_1, \ldots, \alpha_n\rangle, l\rangle\rangle \\
\overline{\sigma}\ \gamma & = & \langle \gamma, \overline{\sigma}\rangle & & &
\end{array}$$

Note that in a type τ of the form $\langle \mu, \omega\rangle$ where $\mu = \langle itcv, l_1\rangle$ and $\omega = \langle \overline{\alpha}, l_2\rangle$, l_1 is not necessarily equal to l_2. For example, in `datatype 'a t = T`, our constraint generator constrains `t`'s type to be equal to $\langle\langle \gamma, l_1\rangle, \langle\langle \alpha\rangle, l_2\rangle\rangle$ where l_1 is the label associated to `t` and l_2 is the label associated to the sequence `'a`.

Functions and syntactic sugar for our type system's related syntax. We define the accessors on the environments used by our type system as follows: let $tse|_{\mathrm{T}}$ be $tse(0)$ (T stands for Type constructor) and $tse|_{\mathrm{V}}$ be $tse(1)$ (V stands for Value). We now define some syntactic sugar to deal with these environments: we write $tse(vid)$ for $tse|_{\mathrm{V}}(vid)$, $tse(tc)$ for $tse|_{\mathrm{T}}(tc)$, $tse + \Gamma$ for $\langle tse|_{\mathrm{T}}, tse|_{\mathrm{V}} + \Gamma\rangle$, $tse + \nabla$ for $\langle tse|_{\mathrm{T}} + \nabla, tse|_{\mathrm{V}}\rangle$, and $tse + tse'$ for $\langle tse|_{\mathrm{T}} + tse'|_{\mathrm{T}}, tse|_{\mathrm{V}} + tse'|_{\mathrm{V}}\rangle$.

Functions on the environments used by our constraint generator. In addition to $fenv$ (defined in Figure 2), we consider new metavariables ranging over VIdEnv: $benv$ and $cenv$. $genv$ range over $\mathsf{GenEnv} = \mathsf{VIdEnv} \cup \mathsf{TyConEnv}$.

We define the accessors on constraints on identifiers as follows: let $\upsilon|_{\mathrm{T}}$ be $\upsilon(0)$ (T stands for Type), $\upsilon|_{\mathrm{C}}$ be $\upsilon(1)$ (C stands for datatype Constructor) and $\upsilon|_{\mathrm{L}}$ be $\upsilon(2)$ (L stands for Label).

Let emEnv be the empty environment $\langle \varnothing, \varnothing, \varnothing, \varnothing\rangle$.

Let us define some functions on environments and constraints: let $env|_{\mathrm{T}}$ be $env(0)$ (T stands for Type constructors), $env|_{\mathrm{C}}$ be $env(1)$ (C stands for datatype Constructors), $env|_{\mathrm{B}}$ be $env(2)$ (B stands for Bindings or Bound identifiers) and $env|_{\mathrm{F}}$ be $env(3)$ (F stands for Free). Note that $env|_{\mathrm{B}}$ is the environment for identifier occurrences that are binding or bound if they are value variables but might turn out not to be so because discovered to be in fact datatype constructors.

The next four functions define injections from each of our four sub-environments (mappings from identifiers to their constraints) to an environment for constraint generation env. Let $\pi_{\mathrm{T}}(genv)$ be $\langle genv, \varnothing, \varnothing, \varnothing\rangle$, $\pi_{\mathrm{C}}(genv)$ be $\langle \varnothing, genv, \varnothing, \varnothing\rangle$, $\pi_{\mathrm{B}}(genv)$ be $\langle \varnothing, \varnothing, genv, \varnothing\rangle$, and $\pi_{\mathrm{F}}(genv)$ be $\langle \varnothing, \varnothing, \varnothing, genv\rangle$. (The meaning of T, C, B and F are as immediately above.)

Functions and syntactic sugar for our constraints. We write $x_1 \overset{l,\overline{vid}}{=\!=\!=} x_2$ for $x_1 \overset{\{l\},\overline{vid}}{=\!=\!=} x_2$.

The function $\mathrm{getLabCs}$ extracts the labels annotating a set of constraints excluding those inside "let" constraints. Note that no label is extracted from a top-level "let" constraint because such a constraint is not annotated by any label. This function is used by filter to extract labels from the top-level constraints (not nested into a "let" constraint). This extraction is required to decide the order in which the top-level constraints should be dealt with during the unification process. The function $\mathrm{getLabCs}$ is defined as follows:

$$\mathrm{getLabCs}(\overline{c}) = \{l \in \overline{l} \mid x_1 \overset{\overline{l},\overline{vid}}{=\!=\!=} x_2 \in \overline{c}\}$$

The function $\mathrm{getSemCs}$ extracts the semantic constraints from a set of constraints and $\mathrm{getSynCs}$ is the function that extracts the syntactic ones:

$$\begin{array}{lcl}
\mathrm{getSynCs}(\overline{c}) & = & \{c \in \overline{c} \mid c \in \mathsf{CsSyn}\} \\
 & \cup & \{c \mid c \in \mathrm{getSynCs}(\overline{c}_1) \cup \mathrm{getSynCs}(\overline{c}_2) \wedge \textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end} \in \overline{c}\} \\
\mathrm{getSemCs}(\overline{c}) & = & \{c \mid c = (x_1 \overset{\overline{l},\overline{vid}}{=\!=\!=} x_2) \in \overline{c}\} \\
 & \cup & \{\textbf{let } \overline{csb}_1, \overline{csb}_2(\mathrm{getSemCs}(\overline{c}_1)) \textbf{ in } \mathrm{getSemCs}(\overline{c}_2) \textbf{ end} \mid \textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end} \in \overline{c}\}
\end{array}$$

Figure 7. Type system

Labelled expression ($lexp : \langle tse, \sigma \rangle$)

$$\frac{exp : \langle tse, \sigma \rangle}{exp^l : \langle tse, \sigma \rangle}$$

Labelled pattern ($lpat : \langle tse, \sigma \rangle$)

$$\frac{pat : \langle tse, \sigma \rangle}{pat^l : \langle tse, \sigma \rangle}$$

Labelled type ($lty : \langle tse, \sigma \rangle$)

$$\frac{ty : \langle tse, \sigma \rangle}{ty^l : \langle tse, \sigma \rangle}$$

Atomic expression ($atexp : \langle tse, \sigma \rangle$)

$$\frac{}{\iota^l : \langle tse, \text{intty} \rangle} \qquad \frac{}{()^l : \langle tse, \text{unitty} \rangle} \qquad \frac{tse(vid) = \langle \overline{\sigma} \cup \{\sigma\}, st \rangle}{vid^l : \langle tse, \sigma \rangle}$$

$$\frac{n \geq 2 \quad \forall i \in \{1, \ldots, n\}.\ lexp_i : \langle tse, \sigma_i \rangle}{(lexp_1, \ldots, lexp_n)^l : \langle tse, \sigma_1 \times \ldots \times \sigma_n \rangle} \qquad \frac{dec : \langle tse, tse' \rangle \quad lexp : \langle tse + tse', \sigma \rangle}{\text{let}^l\, dec \text{ in } lexp \text{ end} : \langle tse, \sigma \rangle} \qquad \frac{lexp : \langle tse, \sigma \rangle}{(lexp)^l : \langle tse, \sigma \rangle}$$

Expression ($exp : \langle tse, \sigma \rangle$)

$$\frac{exp : \langle tse, \sigma_2 \to \sigma_1 \rangle \quad atexp : \langle tse, \sigma_2 \rangle}{\lceil exp\ atexp \rceil^l : \langle tse, \sigma_1 \rangle} \qquad \frac{match : \langle tse, \sigma \rangle}{\text{fn}^l\ match : \langle tse, \sigma \rangle}$$

Pattern ($pat : \langle tse, \sigma \rangle$)

$$\frac{atpat : \langle tse, \langle \Gamma, \sigma \rangle \rangle \quad tse(vid) = \langle \overline{\sigma} \cup \{\sigma \to \sigma'\}, \text{c} \rangle}{\lceil vid^{l'}\ atpat \rceil^l : \langle tse, \langle \Gamma, \sigma' \rangle \rangle}$$

Match ($match : \langle tse, \sigma \rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ mrule_i : \langle tse, \sigma \rangle}{mrule_1 \mid \ldots \mid mrule_n : \langle tse, \sigma \rangle}$$

Match rule ($mrule : \langle tse, \sigma \rangle$)

$$\frac{lpat : \langle tse, \langle \Gamma, \sigma_1 \rangle \rangle \quad lexp : \langle tse + \Gamma, \sigma_2 \rangle}{lpat \overset{l}{\Rightarrow} lexp : \langle tse, \sigma_1 \to \sigma_2 \rangle}$$

Type sequence ($tyseq : \langle tse, \overline{\sigma} \rangle$)

$$\frac{ty : \langle tse, \sigma \rangle}{ty^l : \langle tse, \langle \sigma \rangle \rangle} \qquad \frac{}{\epsilon^l : \langle tse, \langle\rangle \rangle} \qquad \frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ lty_i : \langle tse, \sigma_i \rangle}{(lty_1, \ldots, lty_n)^l : \langle tse, \langle \sigma_1, \ldots, \sigma_n \rangle \rangle}$$

Type ($ty : \langle tse, \sigma \rangle$)

$$\frac{}{tv^l : \langle tse, tv \rangle} \qquad \frac{}{\text{int}^l : \langle tse, \text{intty} \rangle} \qquad \frac{}{\text{unit}^l : \langle tse, \text{unitty} \rangle} \qquad \frac{tse(tc) = \overline{\alpha}\ \gamma \quad tyseq : \langle tse, \overline{\sigma} \rangle \quad |\overline{\alpha}| = |\overline{\sigma}|}{\lceil tyseq\ tc^{l'} \rceil^l : \langle tse, \overline{\sigma}\ \gamma \rangle}$$

$$\frac{n \geq 2 \quad \forall i \in \{1, \ldots, n\}.\ lty_i : \langle tse, \sigma_i \rangle}{\lceil lty_1 \times \cdots \times lty_n \rceil^l : \langle tse, \sigma_1 \times \ldots \times \sigma_n \rangle} \qquad \frac{\forall i \in \{1, 2\}.\ lty_i : \langle tse, \sigma_i \rangle}{lty_1 \overset{l}{\to} lty_2 : \langle tse, \sigma_1 \to \sigma_2 \rangle} \qquad \frac{lty : \langle tse, \sigma \rangle}{(lty)^l : \langle tse, \sigma \rangle}$$

Atomic pattern ($atpat : \langle tse, \sigma \rangle$)

$$\frac{}{\_ : \langle tse, \langle \varnothing, \sigma \rangle \rangle} \qquad \frac{}{\iota^l : \langle tse, \langle \varnothing, \text{intty} \rangle \rangle} \qquad \frac{}{()^l : \langle tse, \langle \varnothing, \text{unitty} \rangle \rangle} \qquad \frac{n \geq 2 \quad \forall i \in \{1, \ldots, n\}.\ lpat_i : \langle tse, \langle \Gamma_i, \sigma_i \rangle \rangle}{(lpat_1, \ldots, lpat_n)^l : \langle tse, \langle \cup_{i \in \{1, \ldots, n\}} \Gamma_i, \sigma_1 \times \ldots \times \sigma_n \rangle \rangle} \text{ where } \text{dj}(\text{dom}(\Gamma_1), \ldots, \text{dom}(\Gamma_n))$$

$$\frac{tse(vid) = \langle \overline{\sigma} \cup \{\overline{\sigma}\ \gamma\}, \text{c} \rangle}{vid^l : \langle tse, \langle \varnothing, \overline{\sigma}\ \gamma \rangle \rangle} \qquad \frac{tse(vid) = \langle \overline{\sigma}, \text{v} \rangle \text{ or } vid \notin \text{dom}(tse)}{vid^l : \langle tse, \langle \{vid \mapsto \langle \{\sigma\}, \text{v} \rangle\}, \sigma \rangle \rangle} \qquad \frac{lpat : \langle tse, \langle \Gamma, \sigma \rangle \rangle}{(lpat)^l : \langle tse, \langle \Gamma, \sigma \rangle \rangle}$$

Constructor binding ($conbind : \langle tse, \langle \Gamma, \overline{\alpha}\ \gamma \rangle \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ \sigma_i \succeq \overline{\alpha}\ \gamma}{vid^l : \langle tse, \langle \{vid \mapsto \langle \{\sigma_1, \ldots, \sigma_n\}, \text{c} \rangle\}, \overline{\alpha}\ \gamma \rangle \rangle} \qquad \frac{lty : \langle tse, \sigma \rangle \quad \forall i \in \{1, \ldots, n\}.\ \sigma_i \succeq \sigma \to \overline{\alpha}\ \gamma}{(vid^l)^{l_1} \text{ of}^{l_2}\ lty : \langle tse, \langle \{vid \mapsto \langle \{\sigma_1, \ldots, \sigma_n\}, \text{c} \rangle\}, \overline{\alpha}\ \gamma \rangle \rangle}$$

Constructor binding sequence ($conbindseq : \langle tse, \langle \Gamma, \overline{\alpha}\ \gamma \rangle \rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ conbind : \langle tse, \langle \Gamma_i, \overline{\alpha}\ \gamma \rangle \rangle}{conbind_1 \mid \ldots \mid conbind_n : \langle tse, \langle \cup_{i \in \{1, \ldots, n\}} \Gamma_i, \overline{\alpha}\ \gamma \rangle \rangle} \text{ where } \text{dj}(\text{dom}(\Gamma_1), \ldots, \text{dom}(\Gamma_n))$$

Datatype binding ($datbind : \langle tse, \langle \nabla, \Gamma, \gamma \rangle \rangle$)

$$\frac{conbindseq : \langle tse, \langle \Gamma, \overline{\alpha}\ \gamma \rangle \rangle \quad \text{seqOfSeq}(tvseq) = \overline{\alpha}}{tvseq\ tc^{l'} \overset{l}{=} conbindseq : \langle tse, \langle \{tc \mapsto \overline{\alpha}\ \gamma\}, \Gamma, \gamma \rangle \rangle} \text{ where } \text{tyVarSet}(conbindseq) \subseteq \text{tyVarSet}(tvseq)$$

Datatype binding sequence ($datbindseq : \langle tse, tse' \rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ datbind_i : \langle tse, \langle \nabla_i, \Gamma_i, \gamma_i \rangle \rangle}{datbind_1 \text{ and } \cdots \text{ and } datbind_n : \langle tse, \langle \cup_{i \in \{1, \ldots, n\}} \nabla_i, \cup_{i \in \{1, \ldots, n\}} \Gamma_i \rangle \rangle} \text{ where } \text{dj}(\text{dom}(\nabla_1), \ldots, \text{dom}(\nabla_n)) \text{ and } \text{dj}(\text{dom}(\Gamma_1), \ldots, \text{dom}(\Gamma_n)) \text{ and } \text{dj}(\{\gamma_1\}, \ldots, \{\gamma_n\})$$

Value binding ($valbind : \langle tse, \Gamma \rangle$)

$$\frac{lpat : \langle tse, \langle \Gamma, \sigma \rangle \rangle \quad lexp : \langle tse + \Gamma, \sigma \rangle}{\text{rec } lpat \overset{l}{=} lexp : \langle tse, \Gamma \rangle}$$

Declaration ($dec : \langle tse, tse' \rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ valbind : \langle tse, \Gamma_i \rangle}{\text{val } valbind : \langle tse, \varnothing, \Gamma_1 \Cup \cdots \Cup \Gamma_n \rangle \rangle} \qquad \frac{datbindseq : \langle tse + tse'|_{\text{T}}, tse' \rangle}{\text{datatype } datbindseq : \langle tse, tse' \rangle} \text{ where } \overline{\gamma} = \{\gamma \mid \overline{\sigma}\ \gamma \in \text{ran}(tse'|_{\text{T}})\} \text{ and } \text{dj}(\overline{\gamma}, \text{tyNameSet}(tse))$$

The function $\mathrm{moveValVar}$ moves in a "let" constraint the bindings (CsBind) from the set of bindings for which the identifier status is unknown (the first set of bindings in a "let" constraint) to the set of bindings for which the status (either value variable or datatype constructor) is known (the second set of bindings). It is used by $\mathrm{ctxtDepsTrue}$ and defined as follow:

$$\mathrm{moveValVar}(\overline{csb}_1, \overline{csb}_2, \overline{vid}) = \langle \overline{csb}_1 \setminus \overline{csb}, \overline{csb}_2 \cup \overline{csb} \rangle \text{ where } \overline{csb} = \{\langle vid, \alpha_1, \alpha_2, poly, l\rangle \in \overline{csb}_1 \mid vid \in \overline{vid}\}$$

The function $\mathrm{ctxtDepsTrue}$ is needed because a recursive function overrides the status of identifiers. To illustrate the use of this function, let us consider the following example:

```
let datatype t = f
in let val rec g = fn f => f
in let val rec f = fn f => f
in f 1 end end end
```

(Note that this code will raise an exception at runtime due to a quirk in SML's definition, but must still be type checked correctly. Anyway, implementations do not follow standard on dynamic semantics here.) In this piece of code, the first three occurrences of `f` are datatype constructors while the last four ones are value variables. Note that the fourth occurrence of `f` overrides the status of the three following occurrences of `f` in the code. If a binding (csb) has been generated under the context dependency that `f` is value variable (and thus stored into the first set of bindings in a "let" constraint) and it is discovered that `f` is really a value variable because it is the name of a recursive function, then the context dependency under `f` is discarded (the binding is moved into the second set of bindings in the same "let" constraint). The function $\mathrm{ctxtDepsTrue}$ is used by the $\mathrm{restrictRec}$ function presented below and defined as follows:

$$\begin{array}{rcl}
\mathrm{ctxtDepsTrue}(\overline{c}, \overline{vid}) & = & \{x_1 \overset{\overline{l},\varnothing}{=\!=} x_2 \mid x_1 \overset{\overline{l},\varnothing}{=\!=} x_2 \in \overline{c}\} \\
& \cup & \{\textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end} \mid \textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end} \in \overline{c} \\
& & \qquad \wedge\ \overline{c}_1' = \mathrm{ctxtDepsTrue}(\overline{c}_1, \overline{vid}) \\
& & \qquad \wedge\ \overline{c}_2' = \mathrm{ctxtDepsTrue}(\overline{c}_2, \overline{vid}) \\
& & \qquad \wedge\ \mathrm{moveValVar}(\overline{csb}_1, \overline{csb}_2, \overline{vid}) = \langle \overline{csb}_1', \overline{csb}_2' \rangle\}
\end{array}$$

In the previous definition, note the use of $\varnothing$ instead of $\overline{vid}$. This is due to fact that $\mathrm{ctxtDepsTrue}$ does not need to handle the general case because these label sets become non empty during unification (and similarly for $\mathrm{ctxtDepsFalse}$ defined below).

The function $\mathrm{ctxtDepsCsBindFalse}$ removes bindings generated for identifiers with the context dependency that they are value variables, when it is discovered that in fact they are datatype constructors. The function $\mathrm{ctxtDepsCsBindFalse}$ is used by $\mathrm{ctxtDepsFalse}$ and defined as follows:

$$\mathrm{ctxtDepsCsBindFalse}(\overline{csb}, \overline{vid}) = \{\langle vid, \alpha_1, \alpha_2, poly, l\rangle \in \overline{csb} \mid vid \notin \overline{vid}\}$$

The function $\mathrm{ctxtDepsFalse}$ is needed because it may be discovered during the constraint generation process that an identifier bound under the context dependency that it is a value variable is in fact a datatype constructor. For example in `fn x => x`, because under the context dependency that `x` is a value variable the second occurrence of `x` is bound to the first occurrence of `x`, a binding (csb) is generated (under the context dependency that `x` is a value variable). If it is discovered that `x` is a datatype constructor, this binding is discarded. The function $\mathrm{ctxtDepsFalse}$ is used by the $\mathrm{compLet}$ relation presented below and defined as follow:

$$\begin{array}{rcl}
\mathrm{ctxtDepsFalse}(\overline{c}, \overline{vid}) & = & \{x_1 \overset{\overline{l},\varnothing}{=\!=} x_2 \mid x_1 \overset{\overline{l},\varnothing}{=\!=} x_2 \in \overline{c}\} \\
& \cup & \{\mathrm{mul}(\overline{l}, \overline{vid}') \in \overline{c} \mid \mathrm{dj}(\overline{vid}', \overline{vid})\} \\
& \cup & \{\mathrm{con}(\overline{l}, \overline{vid}') \in \overline{c} \mid \mathrm{dj}(\overline{vid}', \overline{vid})\} \\
& \cup & \{\mathrm{inc}(\overline{l}) \mid \mathrm{inc}(\overline{l}) \in \overline{c}\} \\
& \cup & \{\mathrm{app}(\overline{l}) \mid \mathrm{app}(\overline{l}) \in \overline{c}\} \\
& \cup & \{\textbf{let } \overline{csb}_1', \overline{csb}_2(\overline{c}_1') \textbf{ in } \overline{c}_2' \textbf{ end} \mid \textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end} \in \overline{c} \\
& & \qquad \wedge\ \overline{c}_1' = \mathrm{ctxtDepsFalse}(\overline{c}_1, \overline{vid}) \\
& & \qquad \wedge\ \overline{c}_2' = \mathrm{ctxtDepsFalse}(\overline{c}_2, \overline{vid}) \\
& & \qquad \wedge\ \overline{csb}_1' = \mathrm{ctxtDepsCsBindFalse}(\overline{csb}_1, \overline{vid})\}
\end{array}$$

C. Constraint generation

Let us start by presenting the functions and relations used by our constraint generator. Note that we sometimes use relations instead of functions because our constraint generator generates "fresh" type variables but the way of generating these variables is irrelevant.

Composition of environments. The function uenv composes environments. It is inspired by the usual intersection of type environment in intersection type systems and allows performing this intersection on the four type environments contained in an env. Let $\mathrm{uenv}(\overline{env}) = env$ where $env|_{\mathrm{T}} = \uplus_{env \in \overline{env}} env|_{\mathrm{T}}$, $env|_{\mathrm{C}} = \uplus_{env \in \overline{env}} env|_{\mathrm{C}}$, $env|_{\mathrm{B}} = \uplus_{env \in \overline{env}} env|_{\mathrm{B}}$, and $env|_{\mathrm{F}} = \uplus_{env \in \overline{env}} env|_{\mathrm{F}}$.

Arity checking. Given a type constructor, $\mathrm{checkTy}$ checks if all its associated constraints limit its arity to a unique one (it checks if the arity of a type name does not vary in a piece of code). This relation is used by genCs and in our constraint generator by: the rule for let expression and the rule for datatype binding sequences.

$$(\{tc_i \mapsto \overline{v}_i\} \uplus \cdots \uplus \{tc_n \mapsto \overline{v}_n\}, \{\alpha_1\} \uplus \cdots \uplus \{\alpha_n\}) \xrightarrow{\mathrm{checkTy}} \cup_{i \in \{1,\ldots,n\}} \{\alpha_i \overset{v|_{\mathrm{L}},\varnothing}{=\!=} v|_{\mathrm{C}} \mid v \in \overline{v}_i\}$$

Functions taking care of identifier status overriding. The function $\mathrm{getSynt}$ extracts from a set of constraints some syntactic ones that are partially dependent on a particular set of context dependencies. We need this function because, for example, if `x` occurs twice in a

pattern then a multi-occurrence constraint is generated with the context dependency that `x` is a value variable. If it appears that `x` is really a value variable then the program points responsible for this constraint have to be included in the multi-occurrence constraint. It is used in $\mathrm{restrictRec}$ and defined as follow:

$$\mathrm{getSynt}(\overline{c}, \overline{vid}) = \{\mathrm{mul}(\overline{l}, \overline{vid}') \in \overline{c} \mid \neg\mathrm{dj}(\overline{vid}, \overline{vid}')\} \cup \{\mathrm{con}(\overline{l}, \overline{vid}') \in \overline{c} \mid \neg\mathrm{dj}(\overline{vid}, \overline{vid}')\}$$

The function $\mathrm{restrictRec}$ is used to update constraints when an identifier is discovered to definitely be a value variable, which happens only when bound with `val rec`. The context dependencies related to this identifier have then to be discarded. Some constraints such as the multi-occurrence constraints need special care as explained above. The function $\mathrm{restrictRec}$ is defined as follows:

$$\begin{array}{rcl}
\mathrm{restrictRec}(\mathit{fenv}, \overline{c}) & = & \mathrm{ctxtDepsTrue}(\mathrm{getSemCs}(\overline{c}), \mathrm{dom}(\mathit{fenv})) \\
& \cup & (\overline{c}' \setminus \overline{c}'') \\
& \cup & \{\mathrm{mul}(\overline{l} \cup \mathrm{labSet}(\overline{vid}' \lhd \mathit{fenv}), \overline{vid}' \setminus \mathrm{dom}(\mathit{fenv})) \mid \mathrm{mul}(\overline{l}, \overline{vid}') \in \overline{c}''\} \\
& \cup & \{\mathrm{con}(\overline{l} \cup \mathrm{labSet}(\overline{vid}' \lhd \mathit{fenv}), \overline{vid}' \setminus \mathrm{dom}(\mathit{fenv})) \mid \mathrm{con}(\overline{l}, \overline{vid}') \in \overline{c}''\} \\
\multicolumn{3}{r}{\text{where } \overline{c}' = \mathrm{getSynCs}(\overline{c}) \text{ and } \overline{c}'' = \mathrm{getSynt}(\overline{c}', \mathrm{dom}(\mathit{fenv})).}
\end{array}$$

Functions generating syntactic constraints. The function $\mathrm{checkVal}$ checks if an identifier occurs in a pattern both applied and not applied to an argument. Note that in our constraint generator $\upsilon|_{\mathrm{T}} \neq \upsilon|_{\mathrm{C}}$ where $\upsilon \in \overline{\upsilon}_i$ only if the constraint on identifier υ has been generated for vid_i occurring inside a pattern. If the occurrence of vid_i associated to υ is inside an expression then it is always the case that $\upsilon|_{\mathrm{T}} = \upsilon|_{\mathrm{C}}$. This is how we perform our checking only on the identifiers occurring inside a pattern. In the following function, the first argument is always an environment on bound bound identifiers for which we don't know the status ($\mathit{env}|_{\mathrm{B}}$ in env). The function $\mathrm{checkVal}$ is defined as follows:

$$\mathrm{checkVal}(\{\mathit{vid}_1 \mapsto \overline{\upsilon}_1\} \uplus \cdots \uplus \{\mathit{vid}_n \mapsto \overline{\upsilon}_n\}, \overline{c}) = \cup_{i \in \{1,\ldots,n\}} \{\mathrm{app}(\{\upsilon|_{\mathrm{L}}\} \cup \overline{l}) \mid \upsilon \in \overline{\upsilon}_i \wedge \upsilon|_{\mathrm{T}} \neq \upsilon|_{\mathrm{C}} \wedge \mathrm{con}(\overline{l}, \{\mathit{vid}_i\}) \in \overline{c}\}$$

The function $\mathrm{incCons}$ generates syntactic constraints when a type variable in a datatype declaration appears to be free:

$$\mathrm{incCons}(x_1, x_2) = \{\mathrm{inc}(\{l\} \cup \mathrm{labSet}(x_2)) \mid \mathit{tv}^l \in \mathrm{lTyVarSet}(x_1) \wedge \mathit{tv} \notin \mathrm{tyVarSet}(x_2)\}$$

The functions $\mathrm{mulConsI}$ and $\mathrm{mulConsT}$ (I stands for Identifier and T for Type variable) check multi-occurrence errors. In $\mathrm{mulConsI}$ The Boolean is used because, for example, the fact that two datatype constructors are defined in the same datatype declaration with the same name is not context-sensitive (b is then false), whereas the fact that two occurrences of a same identifier occur in a pattern is context-sensitive, it depends on the status of this identifier (b is then true). The functions $\mathrm{mulConsI}$ and $\mathrm{mulConsT}$ are defined as follows:

$$\begin{array}{lcl}
\mathrm{mulConsI}(\mathit{genv}, b) & = & \{\mathrm{mul}(\overline{l}, \overline{vid}) \mid \mathit{id} \in \mathrm{dom}(\mathit{genv}) \wedge \overline{l} \subseteq \mathrm{labSet}(\mathit{genv}(\mathit{id})) \wedge |\overline{l}| = 2 \wedge \overline{vid} = \{\mathit{vid} \mid \mathit{vid} = \mathit{id} \wedge b\}\} \\
\mathrm{mulConsT}(\langle \mathit{ltv}_1, \ldots, \mathit{ltv}_n \rangle) & = & \{\mathrm{mul}(\overline{l}, \varnothing) \mid f = \Cup_{i \in \{1,\ldots,n\}} \mathrm{flatTyVar}(\mathit{ltv}_i) \wedge \mathit{tv} \in \mathrm{dom}(f) \wedge \overline{l} \subseteq \mathrm{labSet}(f(\mathit{tv})) \wedge |\overline{l}| = 2\}
\end{array}$$

Bindings generation. Polymorphic or monomorphic bindings are generated by $\mathrm{bindOne}$. For example, for `let val x = y in (x, x) end`, this function will generate the following bindings : $\langle x, \alpha, \alpha_1, \mathrm{poly}, l\rangle$ and $\langle x, \alpha, \alpha_2, \mathrm{poly}, l\rangle$ where l and α are the label and type variable associated to the first occurrence of `x` and α_1 and α_2 are the types associated to the second and third occurrences of `x` respectively. We handle the case where, for example, an identifier occurs more than once in a pattern. For example, if we have a function such as `fn (x, x) = (true, 2)` then we do not assume that there is only one `x` declared. The function $\mathrm{bindOne}$ is defined as follow:

$$\mathrm{bindOne}(\mathit{id}, \overline{\upsilon}_1, \overline{\upsilon}_2, \mathit{poly}) = \{\langle \mathit{id}, \upsilon_1|_{\mathrm{T}}, \upsilon_2|_{\mathrm{T}}, \mathit{poly}, \upsilon_2|_{\mathrm{L}}\rangle \mid \upsilon_1 \in \overline{\upsilon}_1 \wedge \upsilon_2 \in \overline{\upsilon}_2\}$$

Polymorphic bindings. We create polymorphic binding for a let expression as follows:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{compLet}(\mathit{env}_1, \mathit{env}_2, \overline{c}_1, \mathit{env}_3, \overline{c}_2) = \langle \mathit{env}_4, \overline{c}_4\rangle \text{ where} } \\
\overline{\mathit{csb}}_{\mathrm{F}} & = & \cup_{\mathit{vid} \in \mathrm{dom}(\mathit{env}_2|_{\mathrm{F}})} \mathrm{bindOne}(\mathit{vid}, (\mathit{env}_3|_{\mathrm{F}})^{+\mathit{vid}}(\mathit{vid}), (\mathit{env}_2|_{\mathrm{F}})(\mathit{vid}), \mathrm{poly}) \\
\overline{\mathit{csb}}_{\mathrm{C}} & = & \cup_{\mathit{vid} \in \mathrm{dom}(\mathit{env}_2|_{\mathrm{C}})} \mathrm{bindOne}(\mathit{vid}, (\mathit{env}_3|_{\mathrm{C}} \Cup \mathit{env}_3|_{\mathrm{B}} \Cup \mathit{env}_3|_{\mathrm{F}})^{+\mathit{vid}}(\mathit{vid}), (\mathit{env}_2|_{\mathrm{C}})(\mathit{vid}), \mathrm{poly}) \\
\overline{\mathit{csb}}_{\mathrm{T}} & = & \cup_{\mathit{tc} \in \mathrm{dom}(\mathit{env}_2|_{\mathrm{T}})} \mathrm{bindOne}(\mathit{tc}, (\mathit{env}_3|_{\mathrm{T}})^{+\mathit{tc}}(\mathit{tc}), (\mathit{env}_2|_{\mathrm{T}})(\mathit{tc}), \mathrm{poly}) \\
\overline{\mathit{vid}} & = & \mathrm{dom}(\mathit{env}_2|_{\mathrm{C}} \Cup \mathit{env}_2|_{\mathrm{F}}) \\
\mathit{env}'_3 & = & \langle \mathrm{dom}(\mathit{env}_2|_{\mathrm{T}}) \lhd\!\lhd \mathit{env}_3|_{\mathrm{T}}, \overline{\mathit{vid}} \lhd\!\lhd \mathit{env}_3|_{\mathrm{C}}, \overline{\mathit{vid}} \lhd\!\lhd \mathit{env}_3|_{\mathrm{B}}, \overline{\mathit{vid}} \lhd\!\lhd \mathit{env}_3|_{\mathrm{F}}\rangle \\
\overline{c}'_2 & = & \overline{c}_3 \cup \mathrm{ctxtDepsFalse}(\mathrm{restrictRec}(\mathit{env}_1|_{\mathrm{F}}, \overline{c}_2), \mathrm{dom}(\mathit{env}_2|_{\mathrm{C}})) \cup \mathrm{checkVal}(\mathrm{dom}(\mathit{env}_2|_{\mathrm{F}} \Cup \mathit{env}_2|_{\mathrm{C}}) \lhd \mathit{env}_3|_{\mathrm{B}}, \overline{c}_2) \\
\overline{c}_4 & = & \{\textbf{let } \overline{\mathit{csb}}_{\mathrm{F}}, \overline{\mathit{csb}}_{\mathrm{C}} \cup \overline{\mathit{csb}}_{\mathrm{T}}(\overline{c}_1) \textbf{ in } \overline{c}'_2 \textbf{ end}\} \\
\mathit{env}_4 & = & \mathrm{uenv}(\{\mathit{env}'_3, \mathit{env}_1\})
\end{array}$$

Monomorphic bindings. The function $\mathrm{compMono}$ composes a monomorphic environment with another one. fenv_1 is the environment associated to a pattern and fenv_2 is the environment associated to an expression. This function is used to generate a "let" constraint specifying some monomorphic bindings. It is used by match rules and recursive declarations because in SML, each argument of a function can only be associated a monomorphic type. If `x` is in the pattern part of a function and also occurs twice in the expression part, the type of the three occurrences of `x` will eventually be constrained to be equal during the unification process. We need the Boolean b because in the case of a match rule, the constraints generated during unification will also be annotated by context dependencies on the identifiers dealt with, while in the case of a recursive declaration, the status of an identifier being overwritten, no such context dependency is needed (the identifiers dealt with are guaranteed to be value variables). Depending on this b the generated monomorphic bindings will either be in the first set of bindings (if true) of the generated "let" constraint or in the second one (if false). The function $\mathrm{compMono}$ is defined as follows:

$\mathrm{compMono}(\mathit{fenv}_1, \mathit{fenv}_2, b, \overline{c}_1, \overline{c}_2) = \{\textbf{let } \overline{\mathit{csb}}_1, \overline{\mathit{csb}}_2(\varnothing) \textbf{ in } \overline{c}_1 \cup \overline{c}_2 \textbf{ end}\}$ where
$\overline{\mathit{csb}} = \cup_{\mathit{vid} \in \mathrm{dom}(\mathit{fenv}_1)} \mathrm{bindOne}(\mathit{vid}, {\mathit{fenv}_2}^{+\mathit{vid}}(\mathit{vid}), \mathit{fenv}_1(\mathit{vid}), \mathrm{mono})$
if b then $\langle \overline{\mathit{csb}}_1, \overline{\mathit{csb}}_2 \rangle = \langle \overline{\mathit{csb}}, \varnothing \rangle$ else $\langle \varnothing, \overline{\mathit{csb}} \rangle$

Constraint generation. Our constraint generator has judgements of the form $x \Downarrow \langle y, z, \overline{c} \rangle$ where x is a piece of program syntax, y is either env, ren (where ren is a renaming of our different variables) or $\langle \mathit{env}, \mathit{ren} \rangle$ and z is one of 9 different kinds of packages of information. $\overline{c}$ is the set of type constraints accumulated so far. y and z allow building constraints for non-local connections.

Most of the judgements in our constraint generator are similar to the ones for expressions: $\mathit{exp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle$. It expresses that the pre-typing $\langle \mathit{env}, \alpha, \overline{c} \rangle$ is associated to exp using the relation $\Downarrow$.

Let us now present one of the 45 rules of our constraint generator presented in section C.1 below. This rule is used to derive pre-typings for match rules terms (mrule) and is the only one responsible for the generation of context-dependencies. This rule can be expressed as follows: if we assume that $\langle \mathit{env}_1, \alpha_1, \overline{c}_1 \rangle$ is a pre-typing generated for the pattern lpat and $\langle \mathit{env}_2, \alpha_2, \overline{c}_2 \rangle$ is a pre-typing generated for the expression lexp and that no variable occurs in both pre-typings then we can derive a pre-typing $\langle \mathit{env}, \alpha, \overline{c} \rangle$ for the match rule $\mathit{lpat} \overset{l}{\Rightarrow} \mathit{lexp}$. Let us now explain how we combine, in the case of a match rule, the two former pre-typings to build the latter one. env is the combination of the environments env_1 and env_2 (using $\Cup$) where the environments corresponding to the identifiers bound by the match rule are moved to $\mathit{env}|_{\mathrm{B}}$ so that we do not lose information in case these identifiers appear to be datatype constructors. α is a "fresh" type variable. $\overline{c}$ is the union of $\mathrm{compMono}(\mathit{env}_1|_{\mathrm{F}}, \mathit{env}_2|_{\mathrm{F}}, \mathrm{true}, \overline{c}_1, \overline{c}_2)$ $\mathrm{mulConsI}(\mathit{env}_1|_{\mathrm{F}} \Cup \mathit{env}_1|_{\mathrm{C}}, \mathrm{true})$ and $\{\alpha \overset{l,\varnothing}{=\!=} \alpha_1 \overset{l}{\rightarrow} \alpha_2\}$. If α_1 is the constrained (by $\overline{c}_1$) type of the pattern and α_2 is the constrained (by $\overline{c}_2$) type of the expression then α is the constrained (by $\overline{c}$) type of the match rule. Some context dependencies will eventually be introduced because of what $\mathrm{compMono}$ does. For example, during the unification, the constraint $\alpha_1 \overset{\{l\},\{x\}}{=\!=} \alpha_2$ is generated for the labelled expression $\mathrm{fn}^{l_1}\ x^{l_3\,l_2} \overset{l}{\Rightarrow} x^{l_5\,l_4}$, where α_1 (resp. α_2) is a type variable constrained to be equal to the type of the first (resp. second) occurrence of x.

Finally, at the end of the constraint generation, some extra checkings are performed: we check if all the occurrences of a same type constructor have the same arity (this is also checked while dealing with a "let" constraint) and we check if an identifier occurs both at a datatype constructor and a value variable positions in a pattern. The relation genCs associates a set of constraints to an expression:

$$\mathit{exp} \xrightarrow{\mathrm{genCs}} \overline{c} \cup \overline{c}_1 \cup \overline{c}_2 \iff \mathit{exp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle \wedge (\mathit{env}|_{\mathrm{T}}, \overline{\alpha}) \xrightarrow{\mathrm{checkTy}} \overline{c}_1 \wedge \overline{c}_2 = \mathrm{checkVal}(\mathit{env}|_{\mathrm{B}}, \overline{c}) \wedge \mathrm{djv}(\overline{c}, \overline{\alpha})$$

C.1 Constraint generator

Labelled expression ($\mathit{lexp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle$)

$$\frac{\mathit{exp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle}{\mathit{exp}^l \Downarrow \langle \mathit{env}, \alpha', \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\} \cup \overline{c} \rangle}$$

Labelled pattern ($\mathit{lpat} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle$)

$$\frac{\mathit{pat} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle}{\mathit{pat}^l \Downarrow \langle \mathit{env}, \alpha', \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\} \cup \overline{c} \rangle}$$

Atomic expression ($\mathit{atexp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle$)

$$\frac{}{\iota^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{intty}^l\} \rangle} \qquad \frac{}{()^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{unitty}^l\} \rangle}$$

$$\frac{\mathit{env} = \pi_{\mathrm{F}}(\{\mathit{vid} \mapsto \{\langle \alpha_2, \alpha_2, l \rangle\}\}) \quad \mathrm{djv}(\alpha_1, \alpha_2)}{\mathit{vid}^l \Downarrow \langle \mathit{env}, \alpha_1, \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2\} \rangle} \qquad \frac{\mathit{lexp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle \quad \mathrm{djv}(\overline{c}, \alpha')}{(\mathit{lexp})^l \Downarrow \langle \mathit{env}, \alpha', \overline{c} \cup \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\} \rangle}$$

$$\frac{n \geq 2 \quad \forall i \in \{1, \ldots, n\}.\ \mathit{lexp}_i \Downarrow \langle \mathit{env}_i, \alpha_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{(\mathit{lexp}_1, \ldots, \mathit{lexp}_n)^l \Downarrow \langle \mathrm{uenv}(\{\mathit{env}_1, \ldots, \mathit{env}_n\}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \cup \{\alpha \overset{l,\varnothing}{=\!=} \lceil \alpha_1 \times \cdots \times \alpha_n \rceil^l\} \rangle}$$

$$\frac{\mathit{dec} \Downarrow \langle \mathit{env}_1, \mathit{env}_2, \overline{c}_1 \rangle \quad \mathit{lexp} \Downarrow \langle \mathit{env}_3, \alpha_2, \overline{c}_2 \rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2, \overline{\alpha}, \alpha)}{\mathrm{let}^l\, \mathit{dec}\ \mathrm{in}\ \mathit{lexp}\ \mathrm{end} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \cup \{\alpha \overset{l,\varnothing}{=\!=} \alpha_2\} \rangle}$$

where $(\mathrm{dom}(\mathit{env}_2|_{\mathrm{T}}) \lhd \mathit{env}_3|_{\mathrm{T}}, \overline{\alpha}) \xrightarrow{\mathrm{checkTy}} \overline{c}_3$ and $\langle \mathit{env}, \overline{c} \rangle = \mathrm{compLet}(\mathit{env}_1, \mathit{env}_2, \overline{c}_1, \mathit{env}_3, \overline{c}_2 \cup \overline{c}_3)$

Expression ($\mathit{exp} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \rangle$)

$$\frac{\mathit{exp} \Downarrow \langle \mathit{env}_1, \alpha_1, \overline{c}_1 \rangle \quad \mathit{atexp} \Downarrow \langle \mathit{env}_2, \alpha_2, \overline{c}_2 \rangle}{\lceil \mathit{exp}\ \mathit{atexp} \rceil^l \Downarrow \langle \mathit{env}, \alpha, \overline{c}_1 \cup \overline{c}_2 \cup \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2 \overset{l}{\rightarrow} \alpha\} \rangle}$$

where $\mathrm{djv}(\overline{c}_1, \overline{c}_2, \alpha)$ and $\mathit{env} = \mathrm{uenv}(\{\mathit{env}_1, \mathit{env}_2\})$

$$\frac{\mathit{match} \Downarrow \langle \mathit{env}, \overline{\alpha}, \overline{c} \rangle \quad \mathrm{djv}(\overline{c}, \alpha)}{\mathrm{fn}^l\ \mathit{match} \Downarrow \langle \mathit{env}, \alpha, \overline{c} \cup \{\alpha \overset{l,\varnothing}{=\!=} \alpha' \mid \alpha' \in \overline{\alpha}\} \rangle}$$

Match ($\mathit{match} \Downarrow \langle \mathit{env}, \overline{\alpha}, \overline{c} \rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ \mathit{mrule}_i \Downarrow \langle \mathit{env}_i, \alpha_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n)}{\mathit{mrule}_1 \mid \cdots \mid \mathit{mrule}_n \Downarrow \langle \mathrm{uenv}(\cup_{i \in \{1,\ldots,n\}} \{\mathit{env}_i\}), \{\alpha_1, \ldots, \alpha_n\}, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Match rule ($mrule \Downarrow \langle env, \alpha, \overline{c}\rangle$)

$$\frac{lpat \Downarrow \langle env_1, \alpha_1, \overline{c}_1\rangle \quad lexp \Downarrow \langle env_2, \alpha_2, \overline{c}_2\rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2, \alpha)}{lpat \overset{l}{\Rightarrow} lexp \Downarrow \langle env, \alpha, \overline{c} \cup \{\alpha \overset{l,\varnothing}{=\!=} \alpha_1 \overset{l}{\rightarrow} \alpha_2\}\rangle}$$

where $env|_{\mathrm{T}} = env_1|_{\mathrm{T}} \Cup env_2|_{\mathrm{T}}$
and $env|_{\mathrm{C}} = env_1|_{\mathrm{C}} \Cup env_2|_{\mathrm{C}}$
and $env|_{\mathrm{B}} = env_1|_{\mathrm{B}} \Cup env_2|_{\mathrm{B}} \Cup env_1|_{\mathrm{F}} \Cup (\mathrm{dom}(env_1|_{\mathrm{F}}) \lhd env_2|_{\mathrm{F}})$
and $env|_{\mathrm{F}} = \mathrm{dom}(env_1|_{\mathrm{F}}) \lhd\!\!\cdot\, env_2|_{\mathrm{F}}$
and $\overline{c} = \mathrm{compMono}(env_1|_{\mathrm{F}}, env_2|_{\mathrm{F}}, \mathrm{true}, \overline{c}_1, \overline{c}_2) \cup \mathrm{mulConsI}(env_1|_{\mathrm{F}} \Cup env_1|_{\mathrm{C}}, \mathrm{true})$

Type sequence ($tyseq \Downarrow \langle env, \langle \beta_1, \beta_2\rangle, \overline{c}\rangle$)

$$\frac{ty \Downarrow \langle env, \alpha, \overline{c}\rangle \quad \mathrm{djv}(\overline{c}, \beta, \beta', \alpha')}{ty^l \Downarrow \langle env, \langle \beta, \beta'\rangle, \overline{c} \cup \{\beta \overset{l,\varnothing}{=\!=} \langle\langle\alpha\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle\alpha'\rangle, l\rangle\}\rangle} \qquad \frac{\beta \neq \beta'}{\epsilon^l \Downarrow \langle \mathrm{emEnv}, \langle \beta, \beta'\rangle, \{\beta \overset{l,\varnothing}{=\!=} \langle\langle\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle\rangle, l\rangle\}\rangle}$$

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ lty_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i\rangle}{(lty_1, \ldots, lty_n)^l \Downarrow \langle \mathrm{uenv}(\{env_1, \ldots, env_2\}), \langle \beta, \beta'\rangle, \cup_{i\in\{1,\ldots,n\}} \overline{c}_i \cup \{\beta \overset{l,\varnothing}{=\!=} \langle\langle\alpha_1, \ldots, \alpha_n\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle\alpha'_1, \ldots, \alpha'_n\rangle, l\rangle\}\rangle}$$

where $\forall i \in \{1, \ldots, n\}.\ \overline{var}'_i = \mathrm{varSet}(lty_i)$ and $\overline{var}_i = \mathrm{varSet}(\overline{c}_i) \setminus \overline{var}'_i$
and $\mathrm{djv}(\overline{var}_1, \ldots, \overline{var}_n, \cup_{i\in\{1,\ldots,n\}} \overline{var}'_i, \beta, \beta', \alpha'_1, \ldots, \alpha'_n)$

Labelled type constructor ($ltc \Downarrow \langle env, \langle \alpha, \beta, \beta'\rangle, \overline{c}\rangle$)

$$\frac{\mathrm{djv}(\alpha, \alpha', \beta, \beta', \delta, \delta')}{tc^l \Downarrow \langle \pi_{\mathrm{T}}(\{tc \mapsto \{\langle \alpha, \alpha', \varnothing, l\rangle\}\}), \langle \alpha, \beta, \beta'\rangle, \overline{c}\rangle}$$

where $\overline{c} = \{\alpha \overset{l,\varnothing}{=\!=} \langle\langle \delta, l\rangle, \beta\rangle, \alpha' \overset{l,\varnothing}{=\!=} \langle\langle \delta', l\rangle, \beta'\rangle\}$

Labelled type ($lty \Downarrow \langle env, \alpha, \overline{c}\rangle$)

$$\frac{ty \Downarrow \langle env, \alpha, \overline{c}\rangle}{ty^l \Downarrow \langle env, \alpha', \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\} \cup \overline{c}\rangle}$$

Type ($ty \Downarrow \langle env, \alpha, \overline{c}\rangle$)

$$\frac{\alpha \neq tv}{tv^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} tv\}\rangle} \qquad \frac{}{\mathrm{int}^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{intty}^l\}\rangle}$$

$$\frac{}{\mathrm{unit}^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{unitty}^l\}\rangle} \qquad \frac{lty \Downarrow \langle env, \alpha, \overline{c}\rangle \quad \mathrm{djv}(\overline{c}, \alpha')}{(lty)^l \Downarrow \langle env, \alpha', \overline{c} \cup \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\}\rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ lty_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i\rangle}{\lceil lty_1 \times \cdots \times lty_n \rceil^l \Downarrow \langle \mathrm{uenv}(\{env_1, \ldots, env_n\}), \alpha, \overline{c}\rangle}$$

where $\forall i \in \{1, \ldots, n\}.\ \overline{var}_i = \mathrm{varSet}(\overline{c}_i) \setminus \mathrm{varSet}(lty_i)$
and $\mathrm{dj}(\overline{var}_1, \ldots, \overline{var}_n, \{\alpha\}, \cup_{i\in\{1,\ldots,n\}} \mathrm{varSet}(lty_i))$
and $\overline{c} = \cup_{i\in\{1,\ldots,n\}} \overline{c}_i \cup \{\alpha \overset{l,\varnothing}{=\!=} \lceil \alpha_1 \times \cdots \times \alpha_n \rceil^l\}$

$$\frac{\forall i \in \{1, 2\}.\ lty_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i\rangle}{lty_1 \overset{l}{\rightarrow} lty_2 \Downarrow \langle \mathrm{uenv}(\{env_1, env_2\}), \alpha, \overline{c}\rangle}$$

where $\forall i \in \{1, 2\}.\ \overline{var}_i = \mathrm{varSet}(\overline{c}_i) \setminus \mathrm{varSet}(lty_i)$
and $\mathrm{dj}(\overline{var}_1, \overline{var}_2, \{\alpha\}, \cup_{i\in\{1,2\}} \mathrm{varSet}(lty_i))$
and $\overline{c} = \overline{c}_1 \cup \overline{c}_2 \cup \{\alpha \overset{l,\varnothing}{=\!=} \alpha_1 \overset{l}{\rightarrow} \alpha_2\}$

$$\frac{tyseq \Downarrow \langle env_1, \langle \beta_1, \beta_2\rangle, \overline{c}_1\rangle \quad ltc \Downarrow \langle env_2, \langle \alpha, \beta'_1, \beta'_2\rangle, \overline{c}_2\rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2)}{\lceil tyseq\ ltc \rceil^l \Downarrow \langle \mathrm{uenv}(\{env_1, env_2\}), \alpha', \overline{c}\rangle}$$

where $\overline{c} = \overline{c}_1 \cup \overline{c}_2 \cup \{\beta'_1 \overset{l,\varnothing}{=\!=} \beta_1, \alpha' \overset{l,\varnothing}{=\!=} \alpha, \beta'_2 \overset{l,\varnothing}{=\!=} \beta_2\}$

Atomic pattern ($atpat \Downarrow \langle env, \alpha, \overline{c}\rangle$)

$$\frac{}{\_ \Downarrow \langle \mathrm{emEnv}, \alpha, \varnothing\rangle} \qquad \frac{}{\iota^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{intty}^l\}\rangle}$$

$$\frac{}{()^l \Downarrow \langle \mathrm{emEnv}, \alpha, \{\alpha \overset{l,\varnothing}{=\!=} \mathrm{unitty}^l\}\rangle} \qquad \frac{env = \pi_{\mathrm{F}}(\{vid \mapsto \{\langle \alpha_2, \alpha_3, l\rangle\}\}) \quad \mathrm{djv}(\alpha_1, \alpha_2, \alpha_3)}{vid^l \Downarrow \langle env, \alpha_1, \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2, \alpha_3 \overset{l,\varnothing}{=\!=} \langle\langle \delta, l\rangle, \beta\rangle\}\rangle}$$

$$\frac{n \geq 2 \quad \forall i \in \{1, \ldots, n\}.\ lpat_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i\rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{(lpat_1, \ldots, lpat_n)^l \Downarrow \langle \mathrm{uenv}(\{env_1, \ldots, env_n\}), \alpha, \cup_{i\in\{1,\ldots,n\}} \overline{c}_i \cup \{\alpha \overset{l,\varnothing}{=\!=} \lceil \alpha_1 \times \cdots \times \alpha_n \rceil^l\}\rangle}$$

$$\frac{\mathit{lpat} \Downarrow \langle \mathit{env}, \alpha, \overline{c}\rangle \quad \mathrm{djv}(\overline{c}, \alpha')}{(\mathit{lpat})^{l} \Downarrow \langle \mathit{env}, \alpha', \overline{c} \cup \{\alpha' \overset{l,\varnothing}{=\!=} \alpha\}\rangle}$$

Value constructor ($\mathit{lvid} \Downarrow \langle \mathit{env}, \langle \alpha, \alpha', c\rangle, \overline{c}\rangle$)

$$\frac{\mathrm{djv}(\overline{c}, \alpha_1, \alpha_2, \alpha_3, \alpha_4)}{\mathit{vid}^{l} \Downarrow \langle \pi_{\mathrm{C}}(\{\mathit{vid} \mapsto \{\langle \alpha_2, \alpha_4, l\rangle\}\}), \langle \alpha_1, \alpha_3, \mathrm{con}(\{l\}, \{\mathit{vid}\})\rangle, \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2, \alpha_3 \overset{l,\varnothing}{=\!=} \alpha_4, \}\rangle}$$

Pattern ($\mathit{pat} \Downarrow \langle \mathit{env}, \alpha, \overline{c}\rangle$)

$$\frac{\mathit{lvid} \Downarrow \langle \mathit{env}_1, \langle \alpha_1, \alpha_1', c\rangle, \overline{c}_1\rangle \quad \mathit{atpat} \Downarrow \langle \mathit{env}_2, \alpha_2, \overline{c}_2\rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2, \alpha_3, \alpha_4, \alpha_5)}{\lceil \mathit{lvid}\ \mathit{atpat}\rceil^{l} \Downarrow \langle \mathrm{uenv}(\{\mathit{env}_1, \mathit{env}_2\}), \alpha_3, \overline{c}_1 \cup \overline{c}_2 \cup \overline{c}_3 \cup \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha \overset{l}{\rightarrow} \alpha_3, \alpha_1' \overset{l,\varnothing}{=\!=} \alpha_4 \overset{l}{\rightarrow} \alpha_5\}\rangle}$$
where $\overline{c}_3 = \{\mathrm{con}(\overline{l} \cup \{l\}, \overline{\mathit{vid}}) \mid c = \mathrm{con}(\overline{l}, \overline{\mathit{vid}}) \wedge \overline{l} \neq \varnothing\}$

Constructor binding ($\mathit{conbind} \Downarrow \langle \mathit{env}, \alpha, \overline{c}\rangle$)

$$\frac{\mathit{env} = \pi_{\mathrm{C}}(\{\mathit{vid} \mapsto \{\langle \alpha_2, \alpha_2, l\rangle\}\}) \quad \alpha_1 \neq \alpha_2}{\mathit{vid}^{l} \Downarrow \langle \mathit{env}, \alpha_1, \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2, \alpha_2 \overset{l,\varnothing}{=\!=} \langle\langle \delta, l\rangle, \beta\rangle\}\rangle}$$

$$\frac{\mathit{lty} \Downarrow \langle \mathit{env}_1, \alpha, \overline{c}_1\rangle \quad \mathit{lvid} \Downarrow \langle \mathit{env}_2, \langle \alpha_1, \alpha_2, c\rangle, \overline{c}_2\rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2)}{\mathit{lvid}^{l_1}\ \mathrm{of}^{l_2}\ \mathit{lty} \Downarrow \langle \mathrm{uenv}(\{\mathit{env}_1, \mathit{env}_2\}), \alpha_3, \overline{c}'\rangle}$$
where $\overline{c}' = \overline{c} \cup \{\alpha_1 \overset{l_1,\varnothing}{=\!=} \alpha_2, \alpha_2 \overset{l_2,\varnothing}{=\!=} \alpha \overset{l_2}{\rightarrow} \alpha_3\}$

Constructor binding sequence ($\mathit{conbindseq} \Downarrow \langle \mathit{env}, \overline{\alpha}, \overline{c}\rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ \mathit{conbind}_i \Downarrow \langle \mathit{env}_i, \alpha_i, \overline{c}_i\rangle}{\mathit{conbind}_1 \mid \cdots \mid \mathit{conbind}_n \Downarrow \langle \mathrm{uenv}(\{\mathit{env}_1, \ldots, \mathit{env}_n\}), \{\alpha_1, \ldots, \alpha_n\}, \overline{c}\rangle}$$
where $\forall i \in \{1, \ldots, n\}.\ \overline{\mathit{var}}_i = \mathrm{varSet}(\overline{c}_i) \setminus \mathrm{varSet}(\mathit{conbind}_i)$
and $\mathrm{dj}(\overline{\mathit{var}}_1, \ldots, \overline{\mathit{var}}_n, \cup_{i \in \{1,\ldots,n\}} \mathrm{varSet}(\mathit{conbind}_i))$
and $\overline{c} = \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \cup \mathrm{mulConsI}(\uplus_{i \in \{1,\ldots,n\}} \mathit{env}_i|_{\mathrm{C}}, \mathrm{false})$

Labelled type variable ($\mathit{ltv} \Downarrow \langle \mathit{ren}, \alpha, \overline{c}\rangle$)

$$\frac{\mathrm{djv}(\mathit{tv}, \alpha_1, \alpha_2)}{\mathit{tv}^{l} \Downarrow \langle \{\mathit{tv} \mapsto \alpha_2\}, \alpha_1, \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2\}\rangle}$$

Type variable sequence ($\mathit{tvseq} \Downarrow \langle \mathit{ren}, \langle \beta, \beta'\rangle, \overline{c}\rangle$)

$$\frac{\mathit{ltv} \Downarrow \langle \mathit{ren}, \alpha, \overline{c}\rangle \quad \mathrm{djv}(\langle \mathit{ren}, \overline{c}\rangle, \alpha', \beta, \beta')}{\mathit{ltv}^{l} \Downarrow \langle \mathit{ren}, \langle \beta, \beta'\rangle, \{\beta \overset{l,\varnothing}{=\!=} \langle\langle \alpha\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle \alpha'\rangle, l\rangle\}\rangle}$$

$$\frac{\beta \neq \beta'}{\epsilon^{l} \Downarrow \langle \varnothing, \langle \beta, \beta'\rangle, \{\beta \overset{l,\varnothing}{=\!=} \langle\langle\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle\rangle, l\rangle\}\rangle}$$

$$\frac{n \geq 1 \quad \forall i \in \{1, \ldots, n\}.\ \mathit{ltv}_i \Downarrow \langle \mathit{ren}_i, \alpha_i, \overline{c}_i\rangle}{(\mathit{ltv}_1{}^{l_1}, \ldots, \mathit{ltv}_n{}^{l_n})^{l} \Downarrow \langle \mathit{ren}_1 + \ldots + \mathit{ren}_n, \langle \beta, \beta'\rangle, \overline{c} \cup \overline{c}'\rangle}$$
where $\mathrm{djv}(\mathrm{dom}(\mathit{ren} + \ldots + \mathit{ren}_n), \mathrm{ran}(\mathit{ren}_1), \ldots, \mathrm{ran}(\mathit{ren}_n), \alpha_1, \ldots, \alpha_n, \alpha_1', \ldots, \alpha_n', \alpha_1'', \ldots, \alpha_n'', \beta, \beta')$
and $\overline{c} = \cup_{i \in \{1,\ldots,n\}} (\overline{c}_i \cup \{\alpha_i \overset{l_i,\varnothing}{=\!=} \alpha_i''\}) \cup \{\beta \overset{l,\varnothing}{=\!=} \langle\langle \alpha_1'', \ldots, \alpha_n''\rangle, l\rangle, \beta' \overset{l,\varnothing}{=\!=} \langle\langle \alpha_1', \ldots, \alpha_n'\rangle, l\rangle\} \cup \mathrm{mulConsT}(\langle \mathit{ltv}_1, \ldots, \mathit{ltv}_n\rangle)$

Datatype name ($\mathit{datname} \Downarrow \langle\langle \mathit{ren}, \mathit{env}\rangle, \langle \alpha, \beta_1, \beta_1', \beta_2, \beta_2'\rangle, \overline{c}\rangle$)

$$\frac{\mathit{tvseq} \Downarrow \langle \mathit{ren}, \langle \beta_1, \beta_1'\rangle, \overline{c}_1\rangle \quad \mathit{ltc} \Downarrow \langle \mathit{env}, \langle \alpha, \beta_2, \beta_2'\rangle, \overline{c}_2\rangle \quad \mathrm{djv}(\mathrm{dom}(\mathit{ren}), \overline{c}_1, \overline{c}_2)}{\mathit{tvseq}\ \mathit{ltc} \Downarrow \langle\langle \mathit{ren}, \mathit{env}\rangle, \langle \alpha, \beta_1, \beta_1', \beta_2, \beta_2'\rangle, \overline{c}_1 \cup \overline{c}_2\rangle}$$

Datatype binding ($\mathit{datbind} \Downarrow \langle \mathit{env}, \langle \mathit{tenv}, \mathit{cenv}\rangle, \overline{c}\rangle$)

$$\frac{\mathit{datname} \Downarrow \langle\langle \mathit{ren}, \mathit{env}_1\rangle, \langle \alpha, \beta_1, \beta_1', \beta_2, \beta_2'\rangle, \overline{c}_1\rangle \quad \mathit{conbindseq} \Downarrow \langle \mathit{env}_2, \overline{\alpha}, \overline{c}_2\rangle}{\mathit{datname} \overset{l}{=} \mathit{conbindseq} \Downarrow \langle \pi_{\mathrm{T}}((\mathit{env}_2|_{\mathrm{T}})), \langle \mathit{env}_1|_{\mathrm{T}}, \mathit{env}_2|_{\mathrm{C}}\rangle, \overline{c}\rangle}$$
where $\mathrm{djv}(\mathrm{varSet}(\overline{c}_2) \setminus \mathrm{dom}(\mathit{ren}), \mathrm{dom}(\mathit{ren}), \overline{c}_2, \gamma)$
and $\overline{c} = \{\alpha' \overset{l,\varnothing}{=\!=} \alpha \mid \alpha' \in \overline{\alpha}\} \cup \{\alpha \overset{l,\varnothing}{=\!=} \langle\langle \gamma, l\rangle, \beta_1\rangle, \beta_1 \overset{l,\varnothing}{=\!=} \beta_2, \beta_1' \overset{l,\varnothing}{=\!=} \beta_2'\} \cup \overline{c}_1 \cup \overline{c}_2[[\mathit{ren}]] \cup \mathrm{incCons}(\mathit{conbindseq}, \mathit{datname})$

Datatype binding sequence ($\mathit{datbindseq} \Downarrow \langle \mathit{env}, \langle \mathit{tenv}, \mathit{cenv}\rangle, \overline{c}\rangle$)

$$\frac{n \geq 1 \quad \forall i \in \{1,\ldots,n\}.\ datbind_i \Downarrow \langle env_i, (\!|tenv_i, cenv_i|\!), \overline{c}_i\rangle}{datbind_1 \text{ and } \cdots \text{ and } datbind_n \Downarrow \langle env, \langle tenv, \mathrm{uenv}(\{cenv_1,\ldots,cenv_n\})\rangle, \overline{c}\rangle}$$

where $\mathrm{djv}(\overline{c}_1,\ldots,\overline{c}_n,\overline{\alpha})$
and $tenv = \mathrm{uenv}(\{tenv_1,\ldots,tenv_n\})$
and $(\mathrm{dom}(tenv) \lhd \mathrm{uenv}(\{env_1,\ldots,env_n\}), \overline{\alpha}) \xrightarrow{\mathrm{checkTy}} \overline{c}'_1$
and $\langle env, \overline{c}'_2\rangle = \mathrm{compLet}(\varnothing, \pi_{\mathrm{T}}(tenv), \cup_{i\in\{1,\ldots,n\}} \overline{c}_i, \mathrm{uenv}(\{env_1,\ldots,env_n\}), \overline{c}'_1)$
and $\overline{c} = \overline{c}'_2 \cup \mathrm{mulConsI}(\Cup_{i\in\{1,\ldots,n\}} tenv_i, \mathrm{false}) \cup \mathrm{mulConsI}(\Cup_{i\in\{1,\ldots,n\}} cenv_i, \mathrm{false})$

Value binding ($valbind \Downarrow \langle env, fenv, \overline{c}\rangle$)

$$\frac{lpat \Downarrow \langle env_1, \alpha_1, \overline{c}_1\rangle \quad lexp \Downarrow \langle env_2, \alpha_2, \overline{c}_2\rangle \quad \mathrm{djv}(\overline{c}_1, \overline{c}_2, \alpha)}{\text{rec } lpat \overset{l}{=} lexp \Downarrow \langle env, env_1|_{\mathrm{F}}, \mathrm{restrictRec}(env_1|_{\mathrm{F}}, \overline{c} \cup \overline{c}' \cup \{\alpha_1 \overset{l,\varnothing}{=\!=} \alpha_2\})\rangle}$$

where $env|_{\mathrm{T}} = env_1|_{\mathrm{T}} \Cup env_2|_{\mathrm{T}}$
and $env|_{\mathrm{C}} = \mathrm{dom}(env_1|_{\mathrm{F}}) \lhd (env_1|_{\mathrm{C}} \Cup env_2|_{\mathrm{C}})$
and $env|_{\mathrm{B}} = \mathrm{dom}(env_1|_{\mathrm{F}}) \lhd env_2|_{\mathrm{B}}$
and $env|_{\mathrm{F}} = \mathrm{dom}(env_1|_{\mathrm{F}}) \lhd env_2|_{\mathrm{F}}$
and $\overline{c} = \mathrm{mulConsI}(env_1|_{\mathrm{F}} \Cup env_1|_{\mathrm{C}}, \mathrm{true}) \cup \mathrm{checkVal}(\mathrm{dom}(env_1|_{\mathrm{F}}) \lhd env_2|_{\mathrm{B}}, \overline{c}_2)$
and $\overline{c}' = \mathrm{compMono}(env_1|_{\mathrm{F}}, env_2|_{\mathrm{F}}, \mathrm{false}, \overline{c}_1, \overline{c}_2)$

Declarations ($dec \Downarrow \langle env, env', \overline{c}\rangle$)

$$\frac{valbind \Downarrow \langle env, fenv, \overline{c}\rangle}{\text{val } valbind \Downarrow \langle env, \langle \varnothing, \varnothing, \varnothing, fenv\rangle, \overline{c}\rangle} \qquad \frac{datbindseq \Downarrow \langle env, \langle tenv, cenv\rangle, \overline{c}\rangle}{\text{datatype } datbindseq \Downarrow \langle env, \langle tenv, cenv, \varnothing, \varnothing\rangle, \overline{c}\rangle}$$

D. Unification

Syntax. We define the syntactic forms used by our unification algorithm in Figure 8. The mappings mb are used to record the monomorphic bindings during the unification process. They replace the stack used by Pottier and Rémy [23].

Figure 8. Syntactic forms used in our unification algorithm

(Note that err, Error, ek and ErrorKind have already been defined in Figure 3. We merely recall their definitions in this table.)

err	$\in$	Error	$::=$	$\langle \overline{l}, \overline{vid}, ek\rangle$
ρL	$\in$	ExtendedTyLab	$::=$	$\langle \tau, \overline{vid}, \overline{l}\rangle$
ρN	$\in$	ExtendedTyName	$::=$	$\langle \mu, \overline{vid}, \overline{l}\rangle$
ρS	$\in$	ExtendedTySeq	$::=$	$\langle \omega, \overline{vid}, \overline{l}\rangle$
ρ	$\in$	ExtendedTy	$=$	ExtendedTyLab $\cup$ ExtendedTyName $\cup$ ExtendedTySeq
$unitv$	$\in$	UnifierTv	$=$	TyVar $\rightarrow$ ExtendedTyLab
$unitn$	$\in$	UnifierTn	$=$	ITyNameVar $\rightarrow$ ExtendedTyName
$unisq$	$\in$	UnifierSq	$=$	ISeqVar $\rightarrow$ ExtendedTySeq
mb	$\in$	MonoBound	$=$	TyVar $\rightarrow \mathbb{P}$(Label)
uni	$\in$	Unifiers	$::=$	$\langle unitv, unitn, unisq, mb\rangle$
ek	$\in$	ErrorKind	$::=$	$\mathrm{tyConsClash}(\langle l_1, itcv_1\rangle, \langle l_2, itcv_2\rangle)$
			$\mid$	$\mathrm{arityClash}(\langle l_1, n_1\rangle, \langle l_2, n_2\rangle)$
			$\mid$	circularity
			$\mid$	multiOcc
			$\mid$	nonConsApp
			$\mid$	inclusion
			$\mid$	appNotApp
$unifterm$	$\in$	UnifTerm	$::=$	$\mathrm{unify}(uni, \overline{c})$
			$\mid$	$\mathrm{unify}(uni, \overline{c}, \overline{c}', l)$
			$\mid$	$\mathrm{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho}, n, l)$
			$\mid$	$\mathrm{success}(uni)$
			$\mid$	$\mathrm{error}(\overline{l}, \overline{vid}, l, ek)$

End points. The end points of a clashing error (such as type constructor clash or arity clash) are the occurrences of the two clashing pieces of code involved in the error. The precise locations of these end points can be singled out during the unification process using the labels contained into labelled sequence (ω) and labelled names (μ). For example, let us assume that the unification algorithm constrains the labelled name $\langle \mathrm{int}, l_1\rangle$ to be equal to $\langle \mathrm{unit}, l_2\rangle$. Then because int is not equal to unit, the unification algorithm generates a type constructor clash error specifying the two following end points: int at location l_1 (which is the location responsible for the generation of this occurrence of int) and unit at location l_2 (which is the location responsible for the generation of this occurrence of unit). This can be observed in rules (5), (6) and (8).

Syntactic sugar. Let us defined some syntactic sugar to handle unifiers. If $uni = \langle unitv, unitn, unisq, mb\rangle$ then let:

$$
\begin{array}{lll}
\mathrm{dom}(uni) & \text{be} & \mathrm{dom}(unitv) \cup \mathrm{dom}(unitn) \cup \mathrm{dom}(unisq) \\
uni \cup unitv' & \text{be} & \langle unitv \cup unitv', unitn, unisq, mb\rangle \\
uni \cup unitn' & \text{be} & \langle unitv, unitn \cup unitn', unisq, mb\rangle \\
uni \cup unisq' & \text{be} & \langle unitv, unitn, unisq \cup unisq', mb\rangle \\
uni(\alpha) & \text{be} & unitv(\alpha) \\
uni(\delta) & \text{be} & unitn(\delta) \\
uni(\beta) & \text{be} & unisq(\beta) \\
uni \cup mb' & \text{be} & \langle unitv, unitn, unisq, mb \cup mb'\rangle \\
uni \setminus mb' & \text{be} & \langle unitv, unitn, unisq, mb \setminus mb'\rangle
\end{array}
$$

Functions on constraints and types used by the unification algorithm. flatCs is used by the unification algorithm to transform a constraint into an element of one of the unifiers. Let $\mathrm{flatCs}(\alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau) = \alpha \mapsto \langle \tau, \overline{l}, \overline{vid}\rangle$ and $\mathrm{flatCs}(\beta \overset{\overline{l},\overline{vid}}{=\!=} \omega) = \beta \mapsto \langle \omega, \overline{l}, \overline{vid}\rangle$.

$\mathrm{firstCs}$ is used by the unification algorithm to check if the incorporation of a constraint to the current unifiers will generate a circularity problem. Let $\mathrm{firstCs}(\alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau) = \alpha$ and $\mathrm{firstCs}(\beta \overset{\overline{l},\overline{vid}}{=\!=} \omega) = \beta$.

flatTy is used by the unification algorithm to recursively traverse a type structure to check if the addition of a constraint to a unifier would generate a circularity problem. This function forgets the structure of types and instead uses a counter to check if a type contains itself. We define flatTy as follows:

$$
\begin{array}{lcl}
\mathrm{flatTy}(\alpha, \overline{l}, \overline{vid}) & = & \langle\{\langle\alpha, \overline{l}, \overline{vid}\rangle\}, 0\rangle, \\
\mathrm{flatTy}(\langle\mu, \omega\rangle, \overline{l}, \overline{vid}) & = & \mathrm{flatTy}(\omega, \overline{l}, \overline{vid}), \\
\mathrm{flatTy}(\beta, \overline{l}, \overline{vid}) & = & \langle\{\langle\beta, \overline{l}, \overline{vid}\rangle\}, 0\rangle \\
\mathrm{flatTy}(\langle\langle\alpha_1, \ldots, \alpha_n\rangle, l\rangle, \overline{l}, \overline{vid}) & = & \langle\{\langle\alpha_1, \overline{l}, \overline{vid}\rangle, \ldots, \langle\alpha_n, \overline{l}, \overline{vid}\rangle\}, 1\rangle
\end{array}
$$

filter is used by the unification algorithm to filter on labels the set of constraints yet to unify. The unification algorithm works as follows: it selects one label and filters out the constraints that are not annotated by this label. These constraints will be dealt with later. Then, the unification algorithm tries to compute a unifier for the set of constraints annotated by the selected label. It does not matter which label is chosen first. Note that the "let" constraints will always be dealt with at the end of the unification process. filter is defined as follows: $\mathrm{filter}(\overline{c}, l) = \{c \in \overline{c} \mid l \notin \mathrm{getLabCs}(\{c\})\}$.

Building of a type from unifiers. $\mathrm{buildTy}$ is used by the unification algorithm while dealing with a "let" constraint. It is more precisely used by link. This idea is that for a polymorphic binding, the unification algorithm will first unify the constraints associated to the declaration of the bound identifier currently dealt with and from the generated unifier build the type of the identifier. This type will then be duplicated as many times as there are occurrences of the bound identifier in its scope. It avoids duplicating the whole set of unsolved constraints associated to a declaration. $\mathrm{buildTy}$ recursively reconstructs a type using a unifier as follows:

$$
\begin{array}{lcl}
\mathrm{buildTy}(\alpha, uni) & = & \langle\tau, \overline{l} \cup \overline{l}', \overline{vid}\rangle \\
 & & \text{where if } uni(3)(\alpha) = \overline{l}_0 \text{ then } \overline{l}' = \overline{l}_0 \text{ else } \overline{l}' = \varnothing \\
 & & \text{and if } uni(\alpha) = \langle\tau_1, \overline{l}_1, \overline{vid}_1\rangle \text{ and } \mathrm{buildTy}(\tau_1, uni) = \langle\tau_2, \overline{l}_2, \overline{vid}_2\rangle \\
 & & \quad \text{then } \langle\tau, \overline{l}, \overline{vid}\rangle = \langle\tau_2, \overline{l}_1 \cup \overline{l}_2, \overline{vid}_1 \cup \overline{vid}_2\rangle \\
 & & \quad \text{else } \langle\tau, \overline{l}, \overline{vid}\rangle = \langle\alpha, \varnothing, \varnothing\rangle. \\
\mathrm{buildTy}(\langle\mu, \omega\rangle, uni) & = & \langle\langle\mu', \omega'\rangle, \overline{l}_1 \cup \overline{l}_2, \overline{vid}_1 \cup \overline{l}_2\rangle \\
 & & \text{where } \mathrm{buildTy}(\mu, uni) = \langle\mu', \overline{l}_1, \overline{vid}_1\rangle \text{ and } \mathrm{buildTy}(\omega, uni) = \langle\omega', \overline{l}_2, \overline{vid}_2\rangle. \\
\mathrm{buildTy}(\delta, uni) & = & \langle\mu, \overline{l}, \overline{vid}\rangle \\
 & & \text{where if } uni(\delta) = \langle\mu_1, \overline{l}_1, \overline{vid}_1\rangle \text{ and } \mathrm{buildTy}(\mu_1, uni) = \langle\mu_2, \overline{l}_2, \overline{vid}_2\rangle \\
 & & \text{then } \langle\mu, \overline{l}, \overline{vid}\rangle = \langle\mu_2, \overline{l}_1 \cup \overline{l}_2, \overline{vid}_1 \cup \overline{vid}_2\rangle \\
 & & \text{else } \langle\mu, \overline{l}, \overline{vid}\rangle = \langle\delta, \varnothing, \varnothing\rangle \\
\mathrm{buildTy}(itc, uni) & = & \langle itc, \varnothing, \varnothing\rangle \\
\mathrm{buildTy}(\langle itc, l\rangle, uni) & = & \langle\langle itc, l\rangle, \varnothing, \varnothing\rangle \\
\mathrm{buildTy}(\beta, uni) & = & \langle\omega, \overline{l}, \overline{vid}\rangle \\
 & & \text{where if } uni(\beta) = \langle\omega_1, \overline{l}_1, \overline{vid}_1\rangle \text{ and } \mathrm{buildTy}(\omega_1, uni) = \langle\omega_2, \overline{l}_2, \overline{vid}_2\rangle \\
 & & \text{then } \langle\omega, \overline{l}, \overline{vid}\rangle = \langle\omega_2, \overline{l}_1 \cup \overline{l}_2, \overline{vid}_1 \cup \overline{vid}_2\rangle \\
 & & \text{else } \langle\omega, \overline{l}, \overline{vid}\rangle = \langle\beta, \varnothing, \varnothing\rangle \\
\mathrm{buildTy}(\langle\overrightarrow{\tau}, l\rangle, uni) & = & \langle\langle\tau_1, \ldots, \tau_n\rangle, l\rangle, \cup_{i\in\{1,\ldots,n\}} \overline{l}_i, \cup_{i\in\{1,\ldots,n\}} \overline{vid}_i\rangle \\
 & & \text{where } n = |\overrightarrow{\tau}| \text{ and for all } i \in \{1, \ldots, n\}, \mathrm{buildTy}(\overrightarrow{\tau}(i-1), uni) = \langle\tau_i, \overline{l}_i, \overline{vid}_i\rangle.
\end{array}
$$

From bindings to equality constraints. link is used by the unification algorithm to bind an identifier to the occurrences of this identifier in its scope:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{link}(\overline{csb}_1, \overline{csb}_2, uni, \overline{c}) = \langle \overline{c}_1 \cup \overline{c}_2, mb' \rangle \text{ where}} \\
mb & = & \{\alpha \mapsto \overline{l} \mid \alpha' \in \mathrm{dom}(uni(3)) \\
 & & \quad \wedge\ \mathrm{buildTy}(\alpha', uni) = \langle \tau, \overline{l}', \overline{vid} \rangle \\
 & & \quad \wedge\ \alpha \in \mathrm{varSet}(\tau) \setminus \mathrm{dom}(uni(3)) \\
 & & \quad \wedge\ \overline{l} = uni(3)(\alpha') \cup \overline{l}'\} \\
uni_0 & = & uni \cup mb \\
\overline{c}_1 & = & \{\alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau \mid \langle vid, \alpha, \alpha', \mathrm{poly}, l \rangle \in csb_1 \cup csb_2 \\
 & & \quad \wedge\ \mathrm{buildTy}(\alpha', uni_0) = \langle \tau', \overline{l}, \overline{vid} \rangle \\
 & & \quad \wedge\ \mathrm{fresh}(\tau, \tau', \mathrm{dom}(uni_0(3)), \mathrm{varSet}(\overline{c}) \cup \mathrm{varSet}(uni))\} \\
\overline{c}_2 & = & \{\alpha \overset{\{l\},\{vid\}}{=\!=} \alpha' \mid \langle vid, \alpha, \alpha', \mathrm{mono}, l \rangle \in csb_1 \cup csb_2\} \\
mb' & = & \{\alpha' \mapsto \varnothing \mid \alpha \overset{\overline{l},\overline{vid}}{=\!=} \alpha' \in \overline{c}_2\}
\end{array}
$$

The unification algorithm takes as parameter only semantic constraints. It is defined in Figure 9.

E. Minimisation

Informally, the minimisation algorithm works as follows. Let us consider the following piece of code:

```
fun apply2 (f1, _) (x1, _) = (f x, f x)
fun apply1 (f1, f2) (x1, x2) b =
  if b
  then (f1 x1, f2 x2)
  else apply2 (f1, f2) (x1, x2)
fun swap (x1, x2) = (x2, x1)
fun cons x =
  let val g = fn h => h :: x
  in swap (apply1 (g, g) (1, true) false) end
```

Given a type error, the minimisation of this error can be divided into two steps. The first step is a rough minimisation of the found error. It consists in trying to remove the bindings (from the "let" constraints) generated for the different declarations of the untypable piece of code. For the piece of code presented above, the minimisation algorithm first tries to remove the binding generated for `apply2`. This succeeds, because without this binding the piece of code is still untypable (the unification algorithm fails). The labels associated to `apply2`'s declaration can then be removed from the found error. The minimisation algorithm then tries to remove the binding generated for `apply1`. This fails because without this binding the piece of code is typable (the unification algorithm succeeds). Then, the minimisation algorithm tries to remove the bindings generated for the arguments of `apply1`. This fails too. The minimisation algorithm does not try to remove the binding generated for `swap`, because this binding is not originally in the found error. The algorithm then fails in removing the bindings generated for `x`, `g` and `h`.

The second step of the minimisation algorithm consists in trying to remove the different remaining labels from the error once at a time. During this phase, the algorithm builds a new error by finding labels that can be removed from the found error as opposed to its predecessor [9] which was finding labels that cannot be removed from the found error to build a new error from scratch.

This main difference comes from the syntax of the constraints. For example, for the piece of code presented above, the set of generated constraints contains **let** $\varnothing, \overline{csb}(\overline{c}_1)$ **in** $\overline{c}_2$ **end** where $\overline{csb}$ contains the binding generated for the function `apply1` to its occurrence in the last line of the code, $\overline{c}_1$ is the set of constraints associated to `apply1`'s declaration and $\overline{c}_2$ is the set of constraints associated to the scope of `apply1`'s declaration (which is the two following functions in the code). Haack and Wells's minimisation algorithm was using the fact that when failing, the unification algorithm blames a label. This label was then used to build a new error during the minimisation process. In our new unification algorithm, for a polymorphic binding of an identifier, a fresh instance of the type of the identifier is generated for each bound occurrence of the identifier. No specific label (or constraint annotated by this label) in the identifier's declaration cannot then be blamed.

Note that the implementations of our minimisation and enumeration algorithms have an additional minor feature that we do not report on in the present document in order not to complicate the presentation of the algorithms. It consists in memorising a list of filters that have already been tested and led to the success of the unification algorithm. This means that we should never use a superset of such a filter again.

Functions used by our the minimisation algorithm. In order to define our minimisation algorithm, let us first define some useful functions and predicates.

getBindings is used to create a tuple of sets containing all the labels from the bindings of a set of constraints:

$$
\begin{array}{lcl}
\mathrm{getBindings}(\{\textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end}\} \uplus \overline{c},\ \overrightarrow{\overline{l}}) & \rightarrow & \mathrm{getBindings}(\overline{c},\ \overrightarrow{\overline{l}} @ \overrightarrow{\overline{l}}' @ \overrightarrow{\overline{l}}_1 @ \overrightarrow{\overline{l}}_2) \\
 & & \text{where } \overrightarrow{\overline{l}}' = \langle \{l \mid \langle vid, \alpha, \alpha', poly, l \rangle \in \overline{csb}_1 \cup \overline{csb}_2\} \rangle \\
 & & \mathrm{getBindings}(\overline{c}_1, \varnothing) \rightarrow^* \mathrm{getBindings}(\varnothing,\ \overrightarrow{\overline{l}}_1) \\
 & & \mathrm{getBindings}(\overline{c}_2, \varnothing) \rightarrow^* \mathrm{getBindings}(\varnothing,\ \overrightarrow{\overline{l}}_2) \\
\mathrm{getBindings}(\{x_1 \overset{\overline{l},\overline{vid}}{=\!=} x_2\} \uplus \overline{c},\ \overrightarrow{\overline{l}}) & \rightarrow & \mathrm{getBindings}(\overline{c},\ \overrightarrow{\overline{l}})
\end{array}
$$

projLabCs filters out, from a set of constraints, the constraints which are annotated by labels which are in not in the set of labels given as parameter. This function is recursively called when filtering a "let" constraint and also filters out the bindings of such constraints. The function projLabCs is defined as follows:

Figure 9. Unification algorithm

driver rules

(1) $\text{unify}(uni, \varnothing) \rightarrow \text{success}(uni)$

(2) $\text{unify}(uni, \overline{c}) \rightarrow \text{unify}(uni, \overline{c}', \overline{c} \setminus \overline{c}', l)$
if $\overline{c} \neq \varnothing$ and $((l \in \text{getLabCs}(\overline{c})$ and $\overline{c}' = \text{filter}(\overline{c}, l))$ or $(\text{getLabCs}(\overline{c}) = \varnothing$ and $\overline{c}' = \varnothing))$

(3) $\text{unify}(uni, \overline{c}, \varnothing, l) \rightarrow \text{unify}(uni, \overline{c})$

simplification rules

(4) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \mu_1, \omega_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle \mu_2, \omega_2\rangle\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}' \cup \{\mu_1 \overset{\overline{l},\overline{vid}}{=\!=} \mu_2, \omega_1 \overset{\overline{l},\overline{vid}}{=\!=} \omega_2\}, l)$

(5) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle itc_1, l_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle itc_2, l_2\rangle\}, l) \rightarrow \text{error}(\overline{l}, \overline{vid}, l, \text{tyConsClash}(\langle l_1, itc_1\rangle, \langle l_2, itc_2\rangle))$
if $itc_1 \neq itc_2$

(6) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \delta, l_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle itcb, l_2\rangle\}, l) \rightarrow \text{error}(\overline{l}, \overline{vid}, l, \text{tyConsClash}(\langle l_1, \delta\rangle, \langle l_2, itcb\rangle))$

(7) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle itcv, l_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle itcv, l_2\rangle\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}', l)$

(8) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \overline{\alpha}_1, l_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle \overline{\alpha}_2, l_2\rangle\}, l) \rightarrow \text{error}(\overline{l}, \overline{vid}, l, \text{arityClash}(\langle l_1, |\overline{\alpha}_1|\rangle, \langle l_2, |\overline{\alpha}_2|\rangle))$
if $|\overline{\alpha}_1| \neq |\overline{\alpha}_2|$

(9) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \overline{\alpha}_1, l_1\rangle \overset{\overline{l},\overline{vid}}{=\!=} \langle \overline{\alpha}_2, l_2\rangle\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}' \cup \{\overline{\alpha}_1(i) \overset{\overline{l},\overline{vid}}{=\!=} \overline{\alpha}_2(i) \mid i \in \{1, \ldots, n\}\}, l)$
if $n = |\overline{\alpha}_1| = |\overline{\alpha}_2|$

(10) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{var \overset{\overline{l},\overline{vid}}{=\!=} var\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}', l)$

(11) $\text{unify}(uni \cup \{\alpha \mapsto \langle \tau', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \uplus \{\alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau\}, l) \rightarrow \text{unify}(uni \cup \{\alpha \mapsto \langle \tau', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \cup \{\tau' \overset{\overline{l}'',\overline{vid}''}{=\!=\!=} \tau\}, l)$
if $\alpha \neq \tau$ and such that $\overline{l}'' = \overline{l} \cup \overline{l}'$ and $\overline{vid}'' = \overline{vid} \cup \overline{vid}'$

(12) $\text{unify}(uni \cup \{\delta \mapsto \langle \mu', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \uplus \{\langle \delta, l'\rangle \overset{\overline{l},\overline{vid}}{=\!=} \mu\}, l) \rightarrow \text{unify}(uni \cup \{\delta \mapsto \langle \mu', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \cup \{\mu' \overset{\overline{l}'',\overline{vid}''}{=\!=\!=} \mu\}, l)$
if $\mu(0) = \gamma$ or $\mu(0) = \delta' \neq \delta$ and such that $\overline{l}'' = \overline{l} \cup \overline{l}'$ and $\overline{vid}'' = \overline{vid} \cup \overline{vid}'$

(13) $\text{unify}(uni \cup \{\beta \mapsto \langle \omega', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \uplus \{\beta \overset{\overline{l},\overline{vid}}{=\!=} \omega\}, l) \rightarrow \text{unify}(uni \cup \{\beta \mapsto \langle \omega', \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}' \cup \{\omega' \overset{\overline{l}'',\overline{vid}''}{=\!=\!=} \omega\}, l)$
if $\beta \neq \omega$ and such that $\overline{l}'' = \overline{l} \cup \overline{l}'$ and $\overline{vid}'' = \overline{vid} \cup \overline{vid}'$

(14) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau\}, l) \rightarrow \text{occurs}(uni, \overline{c}, \overline{c}', \alpha \overset{\overline{l},\overline{vid}}{=\!=} \tau, \overline{\rho}, n, l)$
if $\alpha \notin \text{dom}(uni) \cup \{\tau\}$ and $\text{flatTy}(\tau, \overline{l}, \overline{vid}) = \langle \overline{\rho}, n\rangle$

(15) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\beta \overset{\overline{l},\overline{vid}}{=\!=} \omega\}, l) \rightarrow \text{occurs}(uni, \overline{c}, \overline{c}', \beta \overset{\overline{l},\overline{vid}}{=\!=} \omega, \overline{\rho}, n, l)$
if $\beta \notin \text{dom}(uni) \cup \{\omega\}$ and $\text{flatTy}(\omega, \overline{l}, \overline{vid}) = \langle \overline{\rho}, n\rangle$

(16) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \delta, l'\rangle \overset{\overline{l},\overline{vid}}{=\!=} \mu\}, l) \rightarrow \text{unify}(uni \cup \{\delta \mapsto \langle \mu, \overline{l}, \overline{vid}\rangle\}, \overline{c}, \overline{c}', l)$
if $\delta \notin \text{dom}(uni)$ and $(\mu(0) = \gamma$ or $\mu(0) = \delta' \neq \delta)$

(17) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \mu, \omega\rangle \overset{\overline{l},\overline{vid}}{=\!=} \alpha\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}' \cup \{\alpha \overset{\overline{l},\overline{vid}}{=\!=} \langle \mu, \omega\rangle\}, l)$

(18) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle \overline{\alpha}, l'\rangle \overset{\overline{l},\overline{vid}}{=\!=} \beta\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}' \cup \{\beta \overset{\overline{l},\overline{vid}}{=\!=} \langle \overline{\alpha}, l'\rangle\}, l)$

(19) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\langle itc, l'\rangle \overset{\overline{l},\overline{vid}}{=\!=} \delta\}, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}' \cup \{\delta \overset{\overline{l},\overline{vid}}{=\!=} \langle itc, l'\rangle\}, l)$

(20) $\text{unify}(uni, \overline{c}, \overline{c}' \uplus \{\textbf{let } \overline{csb}_1, \overline{csb}_2(\overline{c}_1) \textbf{ in } \overline{c}_2 \textbf{ end}\}, l) \rightarrow unifterm$
if $\text{unify}(uni, \overline{c}_1) \rightarrow^* unifterm_1$
and (($unifterm_1 = \text{success}(uni_1)$ and $\text{link}(\overline{csb}_1, \overline{csb}_2, uni_1, \overline{c}) = \langle \overline{c}'', mb\rangle$ and $\text{unify}(uni_1, \overline{c}'') \rightarrow^* unifterm_2$
and (($unifterm_2 = \text{success}(uni_2)$ and $\text{unify}(uni_2 \cup mb, \overline{c}_2) \rightarrow^* unifterm_3$
and (($unifterm_3 = \text{success}(uni_3)$ and $\underline{\text{unify}}(uni_3 \setminus mb, \overline{c}, \overline{c}', l) \rightarrow^* unifterm$)
or $unifterm = unifterm_3 = \text{error}(\overline{l}, \overline{vid}, l', ek)$))
or $unifterm = unifterm_2 = \text{error}(\overline{l}, \overline{vid}, l', ek)$))
or $unifterm = unifterm_1 = \text{error}(\overline{l}, \overline{vid}, l', ek)$)

occurs rules

(21) $\text{occurs}(uni, \overline{c}, \overline{c}', c, \varnothing, n, l) \rightarrow \text{unify}(uni \cup \{\text{flatCs}(c)\}, \overline{c}, \overline{c}', l)$

(22) $\text{occurs}(uni, \overline{c}, \overline{c}', c, \{\langle \text{firstCs}(c), \overline{l}, \overline{vid}\rangle\}, 0, l) \rightarrow \text{unify}(uni, \overline{c}, \overline{c}', l)$

(23) $\text{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho} \uplus \{\langle \text{firstCs}(c), \overline{l}, \overline{vid}\rangle\}, n, l) \rightarrow \text{error}(\overline{l}, \overline{vid}, l, \text{circularity})$
if $n \geq 1$

(24) $\text{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho} \uplus \{\langle \alpha, \overline{l}, \overline{vid}\rangle\}, n, l) \rightarrow \text{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho}, n, l)$
if $\alpha \notin \text{dom}(uni) \cup \{\text{firstCs}(c)\}$

(25) $\text{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho} \uplus \{\langle \beta, \overline{l}, \overline{vid}\rangle\}, n, l) \rightarrow \text{occurs}(uni, \overline{c}, \overline{c}', c, \overline{\rho}, n, l)$
if $\beta \notin \text{dom}(uni) \cup \{\text{firstCs}(c)\}$

(26) $\text{occurs}(uni \cup \{\alpha \mapsto \langle \tau, \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}', c, \overline{\rho} \uplus \{\langle \alpha, \overline{l}, \overline{vid}\rangle\}, n, l) \rightarrow \text{occurs}(uni \cup \{\alpha \mapsto \langle \tau, \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}', c, \overline{\rho} \cup \overline{\rho}', n + n', l)$
if $\text{firstCs}(c) \neq \alpha$ and $\text{flatTy}(\tau, \overline{l} \cup \overline{l}', \overline{vid} \cup \overline{vid}') = \langle \overline{\rho}', n'\rangle$

(27) $\text{occurs}(uni \cup \{\beta \mapsto \langle \omega, \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}', c, \overline{\rho} \uplus \{\langle \beta, \overline{l}, \overline{vid}\rangle\}, n, l) \rightarrow \text{occurs}(uni \cup \{\beta \mapsto \langle \omega, \overline{l}', \overline{vid}'\rangle\}, \overline{c}, \overline{c}', c, \overline{\rho} \cup \overline{\rho}', n + n', l)$
if $\text{firstCs}(c) \neq \beta$ and $\text{flatTy}(\omega, \overline{l} \cup \overline{l}', \overline{vid} \cup \overline{vid}') = \langle \overline{\rho}', n'\rangle$

$$
\begin{array}{lcl}
\mathrm{projLabCs}(\overline{c},\overline{l}) & = & \{x_1 \overset{\overline{l'},\overline{vid}}{=\!=} x_2 \in \overline{c} \mid \overline{l'} \subseteq \overline{l}\} \\
& \cup & \{\textbf{let } \overline{csb'_1}, \overline{csb'_2}(\overline{c'_1}) \textbf{ in } \overline{c'_3} \textbf{ end} \mid \textbf{ let } \overline{csb_1}, \overline{csb_2}(\overline{c_1}) \textbf{ in } \overline{c_3} \textbf{ end} \in \overline{c} \\
& & \qquad \wedge\ \overline{c'_1} = \mathrm{projLabCs}(\overline{c_1},\overline{l}) \\
& & \qquad \wedge\ \overline{c'_2} = \mathrm{projLabCs}(\overline{c_2},\overline{l}) \\
& & \qquad \wedge\ \overline{csb'_1} = \mathrm{projLabCs}(\overline{csb_1},\overline{l}) \\
& & \qquad \wedge\ \overline{csb'_2} = \mathrm{projLabCs}(\overline{csb_2},\overline{l})\} \\
\mathrm{projLabCs}(\overline{csb},\overline{l}) & = & \{\langle vid, \alpha_1, \alpha_2, poly, l\rangle \in \overline{csb} \mid l \in \overline{l}\}
\end{array}
$$

The relation getLastUnifier is used to obtain the unifier (of the form $\langle unitv, unitn, unisq, mb\rangle$) built when unifying a set of constraints:

$$
\begin{array}{lcl}
\overline{c} \xrightarrow{\mathrm{getLastUnifier}} uni & \iff & \mathrm{unify}(\varnothing,\overline{c}) \rightarrow^* \mathrm{success}(uni) \\
& \vee & \mathrm{unify}(\varnothing,\overline{c}) \rightarrow^* \mathrm{unify}(uni,\overline{c},\overline{c}',l) \rightarrow \mathrm{error}(\overline{l},\overline{vid},l,ek) \\
& \vee & \mathrm{unify}(\varnothing,\overline{c}) \rightarrow^* \mathrm{occurs}(uni,\overline{c},\overline{c}',c,\overline{\rho},n,l) \rightarrow \mathrm{error}(\overline{l},\overline{vid},l,ek).
\end{array}
$$

Let $\mathrm{isSuccess}(\overline{c})$ be true iff $\mathrm{unify}(\varnothing,\overline{c}) \rightarrow^* \mathrm{success}(uni)$.

The "unbinding" phase. Let us formally define the two phases of our minimisation algorithm. We call the first one "unbinding" and the second one "reduce". The first phase is called "unbinding" because it tries to remove declarations from a found error. We present this algorithm in Figure 10.

Figure 10. The "unbinding" algorithm

$$
\begin{array}{lll}
\text{(MU1)} & \mathrm{unbind}(\varnothing,\overline{l}_1,\overline{l}_2,\overline{c}) & \rightarrow \quad \mathrm{Unbind}(\overline{l}_2) \\
\text{(MU2)} & \mathrm{unbind}(\langle\overline{l}\rangle@\overrightarrow{\overline{l}},\overline{l}_1,\overline{l}_2,\overline{c}) & \rightarrow \quad \mathrm{unbind}(\overrightarrow{\overline{l}},\overline{l}_1,\overline{l}'_2,\overline{c}) \\
& \text{if (} \quad (\mathrm{isSuccess}(\mathrm{projLabCs}(\overline{c},\overline{l}_1 \setminus (\overline{l}_2 \cup \overline{l}))) \text{ and } \overline{l}'_2 = \overline{l}_2) & \\
& \quad \text{or (not } \mathrm{isSuccess}(\mathrm{projLabCs}(\overline{c},\overline{l}_1 \setminus (\overline{l}_2 \cup \overline{l}))) \text{ and } \overline{l}'_2 = \overline{l}_2 \cup \overline{l}) &
\end{array}
$$

The "reduce" phase. The second phase consists in trying to remove one by one the labels from an enumerated error. We present this algorithm in Figure 11.

Figure 11. The "reduce" algorithm

$$
\begin{array}{lll}
\text{(MR1)} & \mathrm{reduce}(\overline{c},\overline{l},\varnothing,err) & \rightarrow \quad \mathrm{Reduce}(err) \\
\text{(MR2)} & \mathrm{reduce}(\overline{c},\overline{l}_1,\{l\}\cup\overline{l}_2,err) & \rightarrow \quad \mathrm{reduce}(\overline{c},\overline{l}'_1,\overline{l}_2,err') \\
& \text{if (} \quad (\mathrm{unify}(\varnothing,\mathrm{projLabCs}(\overline{c},\overline{l}_1\cup\overline{l}_2)) \rightarrow^* \mathrm{error}(\overline{l}',\overline{vid},l',ek) \text{ and } err' = \langle\overline{l}',\overline{vid},ek\rangle \text{ and } \overline{l}'_1 = \overline{l}_1) & \\
& \quad \text{or } (\mathrm{unify}(\varnothing,\mathrm{projLabCs}(\overline{c},\overline{l}_1\cup\overline{l}_2)) \rightarrow^* \mathrm{success}(uni) \text{ and } err' = err \text{ and } \overline{l}'_1 = \{l\}\cup\overline{l}_1)) &
\end{array}
$$

Minimisation algorithm. Finally the minimisation algorithm makes use of the "unbinding" and "reduce" algorithms. We present this algorithm in Figure 12.

Figure 12. Minimisation algorithm

$$
\begin{array}{lll}
\text{(M)} & \mathrm{minimise}(err,\overline{c}) & \rightarrow \quad \mathrm{Min}(err') \\
& \text{if } err = \langle\overline{l},\overline{vid},ek\rangle & \\
& \text{and } \mathrm{getBindings}(\overline{c},\varnothing) \rightarrow^* \mathrm{getBindings}(\varnothing,\overrightarrow{\overline{l}}) & \\
& \text{and } \mathrm{unbind}(\overrightarrow{\overline{l}},\overline{l},\varnothing,\overline{c}) \rightarrow \mathrm{Unbind}(\overline{l}') & \\
& \text{and } \mathrm{reduce}(\overline{c},\varnothing,\overline{l}\setminus\overline{l}',err) \rightarrow \mathrm{Reduce}(err') &
\end{array}
$$

F. Enumeration

We define the terms of our rewriting system which enumerates the minimal type errors from a set of constraints as follows:

$$
\begin{array}{lcll}
enumterm & \in & \mathsf{EnumTerm} & ::= \quad \mathrm{enum}(uni,\overline{c}) \\
& & & \mid \quad \mathrm{enumSyn}(uni,\overline{c},\overline{c}',\overline{err},) \\
& & & \mid \quad \mathrm{enumSem}(uni,\overline{c},\overline{err},\overline{\overline{l}}) \\
& & & \mid \quad \mathrm{errorSet}(\overline{err})
\end{array}
$$

enumDone computes a new set of minimal errors from a set of minimal errors an a newly found error.

$$
\begin{array}{lcl}
\langle err,\overline{err},\overline{c}\rangle \xrightarrow{\mathrm{enumDone}} \langle\overline{err}',\overline{l}\rangle & \iff & \text{if there exists } err' \text{ such that } err'(0) \subseteq err(0) \text{ and } err'(1) \subseteq err(1) \\
& & \text{then } \overline{err}' = \overline{err} \text{ and } \overline{l} = \varnothing \\
& & \text{else } \overline{err}' = \overline{err} \cup \{err''\} \text{ and } \overline{l} = err''(0) \text{ where } \mathrm{minimise}(err,\overline{c}) \rightarrow^* \mathrm{Min}(err'').
\end{array}
$$

The enumeration algorithm presented in Figure 13 is an extension of the enumeration algorithm introduced by Haack and Wells [9] to cope with our new constraints. As explained before, it also makes use of the minimisation algorithm.

Figure 13. Enumeration algorithm

(E1)	$\text{enum}(uni, \overline{c})$	$\rightarrow$	$\text{enumSyn}(uni, \text{getSynCs}(\overline{c}), \text{getSemCs}(\overline{c}), \varnothing, \varnothing)$
(E2)	$\text{enumSyn}(uni, \varnothing, \overline{c}, \overline{err}, \varnothing)$	$\rightarrow$	$\text{errorSet}(\overline{err})$ if $\text{unify}(uni, \overline{c}) \rightarrow^* \text{success}(uni')$
(E3)	$\text{enumSyn}(uni, \varnothing, \overline{c}, \overline{err}, \varnothing)$	$\rightarrow$	$\text{enumSem}(uni, \overline{c}, \overline{err}', \{\{l'\} \mid l' \in \overline{l}'\})$ if $\text{unify}(uni, \overline{c}) \rightarrow^* \text{error}(\overline{l}, \overline{vid}, ek, l)$ and where $\langle\langle \overline{l}, \overline{vid}, ek\rangle, \overline{err}, \overline{c}\rangle \xrightarrow{\text{enumDone}} \langle \overline{err}', \overline{l}'\rangle$
(E4)	$\text{enumSyn}(uni, \varnothing, \overline{c}, \overline{err}, \overline{l} \cup \{l\})$	$\rightarrow$	$\text{enumSem}(uni, \overline{c}, \overline{err}, \overline{l} \cup \{l\})$
(E5)	$\text{enumSyn}(uni, \overline{c} \uplus \{\text{mul}(\overline{l}, \overline{vid})\}, \overline{c}', \overline{err}, \overline{l}')$	$\rightarrow$	$\text{enumSyn}(uni, \overline{c}, \overline{c}', \overline{err}', \overline{l}' \cup \overline{l}'')$ where $\langle\langle \overline{l}, \overline{vid}, \text{multiOcc}\rangle, \overline{err}, \varnothing\rangle \xrightarrow{\text{enumDone}} \langle \overline{err}', \overline{l}''\rangle$
(E6)	$\text{enumSyn}(uni, \overline{c} \uplus \{\text{con}(\overline{l}, \overline{vid})\}, \overline{c}', \overline{err}, \overline{l}')$	$\rightarrow$	$\text{enumSyn}(uni, \overline{c}, \overline{c}', \overline{err}', \overline{l}' \cup \overline{l}'')$ where $\langle\langle \overline{l}, \overline{vid}, \text{nonConsApp}\rangle, \overline{err}, \varnothing\rangle \xrightarrow{\text{enumDone}} \langle \overline{err}', \overline{l}''\rangle$
(E7)	$\text{enumSyn}(uni, \overline{c} \uplus \{\text{inc}(\overline{l})\}, \overline{c}', \overline{err}, \overline{l}')$	$\rightarrow$	$\text{enumSyn}(uni, \overline{c}, \overline{c}', \overline{err}', \overline{l}' \cup \overline{l}'')$ where $\langle\langle \overline{l}, \varnothing, \text{inclusion}\rangle, \overline{err}, \varnothing\rangle \xrightarrow{\text{enumDone}} \langle \overline{err}', \overline{l}''\rangle$
(E8)	$\text{enumSem}(uni, \overline{c}, \overline{err}, \varnothing)$	$\rightarrow$	$\text{errorSet}(\overline{err})$
(E9)	$\text{enumSem}(uni, \overline{c}, \overline{err}, \overline{\overline{l}} \uplus \{\overline{l}\})$	$\rightarrow$	$\text{enumSem}(uni, \overline{c}, \overline{err}', \overline{\overline{l}}')$, if $\text{unify}(uni, \text{filter}(\overline{c}, \overline{l})) \rightarrow^* \text{error}(\overline{l}', \overline{vid}, ek, l)$ and $\langle\langle \overline{l}', \overline{vid}, ek\rangle, \overline{err}, \overline{c}\rangle \xrightarrow{\text{enumDone}} \langle \overline{err}', \overline{l}''\rangle$ and $\overline{\overline{l}}' = \overline{\overline{l}} \cup \{\overline{l} \cup \{l'\} \mid l' \in \overline{l}'' \wedge \forall \overline{l}_0 \in \overline{\overline{l}}.\ \overline{l}_0 \not\subseteq \overline{l} \cup \{l'\}\}$
(E10)	$\text{enumSem}(uni, \overline{c}, \overline{err}, \overline{\overline{l}} \uplus \{\overline{l}\})$	$\rightarrow$	$\text{enumSem}(uni, \overline{c}, \overline{err}, \overline{\overline{l}})$, if $\text{unify}(uni, \text{filter}(\overline{c}, \overline{l})) \rightarrow^* \text{success}(uni')$

G. Slicing

Syntax. We extend the syntax presented in Figure 5 with "dot" terms as follows:

$part$	$\in$	Part	::=	$pat \mid exp \mid dec \mid ty \mid tyseq$
$lvid$	$\in$	LabId	::=	$\ldots \mid \text{dots}(\varnothing)$
ltc	$\in$	LabTyCon	::=	$\ldots \mid \text{dots}(\varnothing)$
ltv	$\in$	LabTyVar	::=	$\ldots \mid \text{dots}(\varnothing)$
$llvid$	$\in$	LabLabVId	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$tvseq$	$\in$	TyVarSeq	::=	$\ldots \mid \text{dots}(\overrightarrow{ltv})$
lty	$\in$	LabTy	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
ty	$\in$	Ty	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$tyseq$	$\in$	TySeq	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$conbind$	$\in$	ConBind	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$conbindseq$	$\in$	ConBindSeq	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$valbind$	$\in$	ValBind	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$datname$	$\in$	DatName	::=	$\ldots \mid \text{dots}(\varnothing)$
$datbind$	$\in$	DatBind	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$datbindseq$	$\in$	DatBindSeq	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
dec	$\in$	Dec	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$atexp$	$\in$	AtExp	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$lexp$	$\in$	LabExp	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
exp	$\in$	Exp	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$match$	$\in$	Match	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$mrule$	$\in$	MRule	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$atpat$	$\in$	AtPat	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
$lpat$	$\in$	LabPat	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$
pat	$\in$	Pat	::=	$\ldots \mid \text{dots}(\overrightarrow{part})$

Flattening. We now provide the definitions of our different flattening functions.

First let us define some syntactic forms:

$some$	$\in$	Some	::=	$part \mid lexp \mid lty \mid lpat \mid lltv \mid llvid$
seq	$\in$	Seq	::=	$mrule \mid conbind \mid datbind$

Our first flattening function flatten is for non sequential terms:

$$
\begin{array}{lcl}
\mathrm{flatten}(\varnothing) & = & \varnothing \\
\mathrm{flatten}(\langle some \rangle @ \overrightarrow{some}) & = & \begin{cases}
\overrightarrow{part} @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = \mathrm{dots}(\overrightarrow{part}) \\
\langle exp \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = exp^{l} \\
\langle ty \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = ty^{l} \\
\langle pat \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = pat^{l} \\
\langle ltv \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = ltv^{l} \\
\langle lvid \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{if } some = lvid^{l} \\
\langle some \rangle @ \mathrm{flatten}(\overrightarrow{some}) & \text{otherwise}
\end{cases}
\end{array}
$$

Our second flattening function $\mathrm{seqFlatten}$ is for sequential terms:

$$
\begin{array}{lcl}
\mathrm{seqFlatten}(\varnothing) & = & \varnothing \\
\mathrm{seqFlatten}(\langle seq \rangle @ \overrightarrow{seq}) & = & \begin{cases}
\mathrm{seqFlatten}(\mathrm{dots}(\overrightarrow{part} @ \overrightarrow{part}') @ \overrightarrow{seq}') & \text{if } \overrightarrow{seq} = \langle \mathrm{dots}(\overrightarrow{part}') \rangle @ \overrightarrow{seq}' \\
\mathrm{seqFlatten}(\overrightarrow{seq}) & \text{if } \overrightarrow{part} = \varnothing \\
\langle seq \rangle @ \mathrm{seqFlatten}(\overrightarrow{seq}) & \text{otherwise}
\end{cases} \\
 & & \text{where } seq = \mathrm{dots}(\overrightarrow{part}) \\
\mathrm{seqFlatten}(\langle seq \rangle @ \overrightarrow{seq}) & = & \langle seq \rangle @ \mathrm{seqFlatten}(\overrightarrow{seq}) \\
 & & \text{where } seq \neq \mathrm{dots}(\overrightarrow{part})
\end{array}
$$

Type system, constraint generator and slicing algorithm. We extend the function $\mathrm{flatTyVar}$ as follows: let $\mathrm{flatTyVar}(\mathrm{dots}(\varnothing)) = \varnothing$.

We extend our type system in Figure 14, we extend our constraint generator in Figure 15 and we define our slicing algorithm as follows in section G.1. The extension of our constraint generator is the an initial step towards a proof of the correctness of our method (that the minimal slices generated by our type error slicer are untypable and minimal). Given a minimal slice, we aim to prove that the set of constraints generated for the slice is unsolvable and that the slicing of the slice would return the same exact slice.

G.1 Slicing algorithm

Labelled expression ($\langle lexp, \overline{l} \rangle \downarrow lexp'$)

$$\frac{\langle exp, \overline{l} \rangle \downarrow exp' \quad l \in \overline{l}}{\langle exp^{l}, \overline{l} \rangle \downarrow exp'^{l}} \qquad \frac{\langle exp, \overline{l} \rangle \downarrow exp' \quad l \notin \overline{l}}{\langle exp^{l}, \overline{l} \rangle \downarrow \mathrm{flatten}(\langle exp' \rangle)}$$

Atomic expression ($\langle atexp, \overline{l} \rangle \downarrow atexp'$)

$$\frac{x \in \{vid, \iota, ()\}}{\langle x^{l}, \{l\} \cup \overline{l} \rangle \downarrow x^{l}} \qquad \frac{x \in \{vid, \iota, ()\} \quad l \notin \overline{l}}{\langle x^{l}, \overline{l} \rangle \downarrow \mathrm{dots}(\varnothing)}$$

$$\frac{\langle lexp, \overline{l} \rangle \downarrow lexp' \quad l \in \overline{l}}{\langle (lexp)^{l}, \overline{l} \rangle \downarrow (lexp')^{l}} \qquad \frac{\langle lexp, \overline{l} \rangle \downarrow lexp' \quad l \notin \overline{l}}{\langle (lexp)^{l}, \overline{l} \rangle \downarrow \mathrm{dots}(\mathrm{flatten}(\langle lexp' \rangle))}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle lexp_i, \overline{l} \rangle \downarrow lexp'_i \quad l \in \overline{l}}{\langle (lexp_1, \ldots, lexp_n)^{l}, \overline{l} \rangle \downarrow (lexp'_1, \ldots, lexp'_n)^{l}} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ \langle lexp_i, \overline{l} \rangle \downarrow lexp'_i \quad l \notin \overline{l}}{\langle (lexp_1, \ldots, lexp_n)^{l}, \overline{l} \rangle \downarrow \mathrm{dots}(\mathrm{flatten}(\langle lexp'_1, \ldots, lexp'_n \rangle))}$$

$$\frac{\langle dec, \overline{l} \rangle \downarrow dec' \quad \langle lexp, \overline{l} \rangle \downarrow lexp' \quad l \in \overline{l}}{\langle \mathrm{let}^{l}\ dec\ \mathrm{in}\ lexp\ \mathrm{end}, \overline{l} \rangle \downarrow \mathrm{let}^{l}\ dec'\ \mathrm{in}\ lexp'\ \mathrm{end}} \qquad \frac{\langle dec, \overline{l} \rangle \downarrow dec' \quad \langle lexp, \overline{l} \rangle \downarrow lexp' \quad l \notin \overline{l}}{\langle \mathrm{let}^{l}\ dec\ \mathrm{in}\ lexp\ \mathrm{end}, \overline{l} \rangle \downarrow \mathrm{dots}(\mathrm{flatten}(\langle dec', lexp' \rangle))}$$

Expression ($\langle exp, \overline{l} \rangle \downarrow exp'$)

$$\frac{\langle match, \overline{l} \rangle \downarrow match' \quad l \in \overline{l}}{\langle \mathrm{fn}^{l}\ match, \overline{l} \rangle \downarrow \mathrm{fn}^{l}\ match'}$$

$$\frac{\langle match, \overline{l} \rangle \downarrow mrule_1 \mid \cdots \mid mrule_2 \quad l \notin \overline{l}}{\langle \mathrm{fn}^{l}\ match, \overline{l} \rangle \downarrow \mathrm{fn}^{l}\ mrule_1 \mid \cdots \mid mrule_2} \qquad \frac{\langle match, \overline{l} \rangle \downarrow \mathrm{dots}(\overrightarrow{part}) \quad l \notin \overline{l}}{\langle \mathrm{fn}^{l}\ match, \overline{l} \rangle \downarrow \mathrm{dots}(\overrightarrow{part})}$$

$$\frac{\langle lexp, \overline{l} \rangle \downarrow lexp' \quad \langle atexp, \overline{l} \rangle \downarrow atexp' \quad l \in \overline{l}}{\langle \lceil lexp\ atexp \rceil^{l}, \overline{l} \rangle \downarrow \lceil lexp'\ atexp' \rceil^{l}} \qquad \frac{\langle lexp, \overline{l} \rangle \downarrow lexp' \quad \langle atexp, \overline{l} \rangle \downarrow atexp' \quad l \notin \overline{l}}{\langle \lceil lexp\ atexp \rceil^{l}, \overline{l} \rangle \downarrow \mathrm{dots}(\mathrm{flatten}(\langle lexp', atexp' \rangle))}$$

Match ($\langle match, \overline{l} \rangle \downarrow match'$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle mrule_i, \overline{l} \rangle \downarrow mrule'_i \quad \mathrm{seqFlatten}(\langle mrule'_1, \ldots, mrule'_n \rangle) = \langle \mathrm{dots}(\overrightarrow{part}) \rangle}{\langle mrule_1 \mid \cdots \mid mrule_n, \overline{l} \rangle \downarrow \mathrm{dots}(\overrightarrow{part})}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ \langle mrule_i, \overline{l}\rangle \downarrow mrule_i' \quad \text{seqFlatten}(\langle mrule_1', \ldots, mrule_n'\rangle) = \langle mrule_1'', \ldots, mrule_m''\rangle}{\langle mrule_1 \mid \cdots \mid mrule_n, \overline{l}\rangle \downarrow mrule_1'' \mid \cdots \mid mrule_m''}$$

Match rule ($\langle mrule, \overline{l}\rangle \downarrow mrule'$)

$$\frac{\langle lpat, \overline{l}\rangle \downarrow lpat' \quad \langle lexp, \overline{l}\rangle \downarrow lexp' \quad (l \in \overline{l} \text{ or } lpat \neq \text{dots}(\varnothing))}{\langle lpat \stackrel{l}{\Rightarrow} lexp, \overline{l}\rangle \downarrow lpat' \stackrel{l}{\Rightarrow} lexp'} \qquad \frac{\langle lpat, \overline{l}\rangle \downarrow \text{dots}(\varnothing) \quad \langle lexp, \overline{l}\rangle \downarrow lexp' \quad l \notin \overline{l}}{\langle lpat \stackrel{l}{\Rightarrow} lexp, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lexp'\rangle))}$$

Labelled pattern ($\langle lpat, \overline{l}\rangle \downarrow lpat'$)

$$\frac{\langle pat, \overline{l}\rangle \downarrow pat' \quad l \in \overline{l}}{\langle pat^l, \overline{l}\rangle \downarrow pat'^l} \qquad \frac{\langle pat, \overline{l}\rangle \downarrow pat' \quad l \notin \overline{l}}{\langle pat^l, \overline{l}\rangle \downarrow \text{flatten}(\langle pat'\rangle)}$$

Atomic pattern ($\langle atpat, \overline{l}\rangle \downarrow atpat'$)

$$\frac{}{\langle \_, \overline{l}\rangle \downarrow \text{dots}(\varnothing)} \qquad \frac{x \in \{vid, \iota, ()\}}{\langle x^l, \{l\} \cup \overline{l}\rangle \downarrow x^l} \qquad \frac{x \in \{vid, \iota, ()\} \quad l \notin \overline{l}}{\langle x^l, \overline{l}\rangle \downarrow \text{dots}(\varnothing)} \qquad \frac{\langle lpat, \overline{l}\rangle \downarrow lpat' \quad l \notin \overline{l}}{\langle (lpat)^l, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lpat'\rangle))}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ \langle lpat_i, \overline{l}\rangle \downarrow lpat_i' \quad l \in \overline{l}}{\langle (lpat_1, \ldots, lpat_n)^l, \overline{l}\rangle \downarrow (lpat_1', \ldots, lpat_n')^l} \qquad \frac{\forall i \in \{1,\ldots,n\}.\ \langle lpat_i, \overline{l}\rangle \downarrow lpat_i' \quad l \notin \overline{l}}{\langle (lpat_1, \ldots, lpat_n)^l, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lpat_1', \ldots, lpat_n'\rangle))}$$

$$\frac{\langle lpat, \overline{l}\rangle \downarrow lpat' \quad l \in \overline{l}}{\langle (lpat)^l, \overline{l}\rangle \downarrow (lpat')^l}$$

Pattern ($\langle pat, \overline{l}\rangle \downarrow pat'$)

$$\frac{\langle atpat, \overline{l}\rangle \downarrow atpat' \quad \{l_1, l_2\} \subseteq \overline{l}}{\langle \lceil vid^{l_1}\ atpat\rceil^{l_2}, \overline{l}\rangle \downarrow \lceil vid^{l_1}\ atpat'\rceil^{l_2}} \qquad \frac{\langle atpat, \overline{l}\rangle \downarrow atpat' \quad l_1 \notin \overline{l} \quad l_2 \in \overline{l}}{\langle \lceil vid^{l_1}\ atpat\rceil^{l_2}, \overline{l}\rangle \downarrow \lceil \text{dots}(\varnothing)\ atpat'\rceil^{l_2}}$$

$$\frac{\langle atpat, \overline{l}\rangle \downarrow atpat' \quad \text{dj}(\{l_1, l_2\}, \overline{l})}{\langle \lceil vid^{l_1}\ atpat\rceil^{l_2}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle atpat'\rangle))} \qquad \frac{\langle atpat, \overline{l}\rangle \downarrow atpat' \quad l_1 \in \overline{l} \quad l_2 \notin \overline{l}}{\langle \lceil vid^{l_1}\ atpat\rceil^{l_2}, \overline{l}\rangle \downarrow \text{dots}(\langle vid^{l_1}\rangle @ \text{flatten}(\langle atpat'\rangle))}$$

Labelled type variable ($\langle ltv, \overline{l}\rangle \downarrow ltv'$)

$$\frac{l \in \overline{l}}{\langle tv^l, \overline{l}\rangle \downarrow tv^l} \qquad \frac{l \notin \overline{l}}{\langle tv^l, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

Labelled labelled type variable ($\langle lltv, \overline{l}\rangle \downarrow lltv'$)

$$\frac{\langle ltv, \overline{l}\rangle \downarrow ltv' \quad l \in \overline{l}}{\langle ltv^l, \overline{l}\rangle \downarrow ltv'^l} \qquad \frac{\langle ltv, \overline{l}\rangle \downarrow ltv' \quad l \notin \overline{l}}{\langle ltv^l, \overline{l}\rangle \downarrow \text{dots}(\langle ltv'\rangle)}$$

Type variable sequence ($\langle tvseq, \overline{l}\rangle \downarrow tvseq'$)

$$\frac{\langle ltv, \overline{l}\rangle \downarrow ltv' \quad l \in \overline{l}}{\langle ltv^l, \overline{l}\rangle \downarrow ltv'^l}$$

$$\frac{\langle ltv, \overline{l}\rangle \downarrow tv^{l'} \quad l \notin \overline{l}}{\langle ltv^l, \overline{l}\rangle \downarrow \text{dots}(\langle tv^{l'}\rangle)} \qquad \frac{\langle ltv, \overline{l}\rangle \downarrow \text{dots}(\varnothing) \quad l \notin \overline{l}}{\langle ltv^l, \overline{l}\rangle \downarrow \text{dots}(\varnothing)} \qquad \frac{l \in \overline{l}}{\langle \epsilon^l, \overline{l}\rangle \downarrow \epsilon^l} \qquad \frac{l \notin \overline{l}}{\langle \epsilon^l, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ \langle lltv_i, \overline{l}\rangle \downarrow lltv_i' \quad l \in \overline{l}}{\langle (lltv_1, \ldots, lltv_n)^l, \overline{l}\rangle \downarrow (lltv_1', \ldots, lltv_n')^l} \qquad \frac{\forall i \in \{1,\ldots,n\}.\ \langle lltv_i, \overline{l}\rangle \downarrow lltv_i' \quad l \notin \overline{l}}{\langle (lltv_1, \ldots, lltv_n)^l, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\text{flatten}(\langle lltv_1', \ldots, lltv_n'\rangle)))}$$

Labelled type ($\langle lty, \overline{l}\rangle \downarrow lty'$)

$$\frac{\langle ty, \overline{l}\rangle \downarrow ty' \quad l \in \overline{l}}{\langle ty^{l}, \overline{l}\rangle \downarrow ty'^{l}} \qquad \frac{\langle ty, \overline{l}\rangle \downarrow ty' \quad l \notin \overline{l}}{\langle ty^{l}, \overline{l}\rangle \downarrow \text{flatten}(\langle ty'\rangle)}$$

Type ($\langle ty, \overline{l}\rangle \downarrow ty'$)

$$\frac{l \in \overline{l}}{\langle \text{int}^{l}, \overline{l}\rangle \downarrow \text{int}^{l}} \qquad \frac{l \notin \overline{l}}{\langle \text{int}^{l}, \overline{l}\rangle \downarrow \text{dots}(\varnothing)} \qquad \frac{l \in \overline{l}}{\langle \text{unit}^{l}, \overline{l}\rangle \downarrow \text{unit}^{l}} \qquad \frac{l \notin \overline{l}}{\langle \text{unit}^{l}, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

$$\frac{\langle lty_1, \overline{l}\rangle \downarrow lty_1' \quad \langle lty_2, \overline{l}\rangle \downarrow lty_2' \quad l \in \overline{l}}{\langle lty_1 \xrightarrow{l} lty_2, \overline{l}\rangle \downarrow lty_1' \xrightarrow{l} lty_2'} \qquad \frac{\langle lty_1, \overline{l}\rangle \downarrow lty_1' \quad \langle lty_2, \overline{l}\rangle \downarrow lty_2' \quad l \notin \overline{l}}{\langle lty_1 \xrightarrow{l} lty_2, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lty_1', lty_2'\rangle))}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle lty_i, \overline{l}\rangle \downarrow lty_i' \quad l \in \overline{l}}{\langle \lceil lty_1 \times \cdots \times lty_n \rceil^{l}, \overline{l}\rangle \downarrow \lceil lty_1' \times \cdots \times lty_n' \rceil^{l}} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ \langle lty_i, \overline{l}\rangle \downarrow lty_i' \quad l \notin \overline{l}}{\langle \lceil lty_1 \times \cdots \times lty_n \rceil^{l}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lty_1', \ldots, lty_n'\rangle))}$$

$$\frac{\langle tyseq, \overline{l}\rangle \downarrow tyseq' \quad (l_1 \in \overline{l} \text{ or } l_2 \in \overline{l})}{\langle \lceil tyseq\ tc^{l_1} \rceil^{l_2}, \overline{l}\rangle \downarrow \lceil tyseq'\ tc^{l_1} \rceil^{l_2}} \qquad \frac{\langle tyseq, \overline{l}\rangle \downarrow tyseq' \quad l_1 \notin \overline{l} \quad l_2 \notin \overline{l}}{\langle \lceil tyseq\ tc^{l_1} \rceil^{l_2}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle tyseq'\rangle))}$$

$$\frac{\langle lty, \overline{l}\rangle \downarrow lty' \quad l \in \overline{l}}{\langle (lty)^{l}, \overline{l}\rangle \downarrow (lty')^{l}} \qquad \frac{\langle lty, \overline{l}\rangle \downarrow lty' \quad l \notin \overline{l}}{\langle (lty)^{l}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lty'\rangle))}$$

Type Sequence ($\langle tyseq, \overline{l}\rangle \downarrow tyseq'$)

$$\frac{\langle ty, \overline{l}\rangle \downarrow ty' \quad l \in \overline{l}}{\langle ty^{l}, \overline{l}\rangle \downarrow ty'^{l}} \qquad \frac{\langle ty, \overline{l}\rangle \downarrow ty' \quad l \notin \overline{l}}{\langle ty^{l}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle ty'\rangle))} \qquad \frac{l \in \overline{l}}{\langle \epsilon^{l}, \overline{l}\rangle \downarrow \epsilon^{l}} \qquad \frac{l \notin \overline{l}}{\langle \epsilon^{l}, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle lty_i, \overline{l}\rangle \downarrow lty_i' \quad l \in \overline{l}}{\langle (lty_1, \ldots, lty_n)^{l}, \overline{l}\rangle \downarrow (lty_1', \ldots, lty_n')^{l}} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ \langle lty_i, \overline{l}\rangle \downarrow lty_i' \quad l \notin \overline{l}}{\langle (lty_1, \ldots, lty_n)^{l}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lty_1', \ldots, lty_n'\rangle))}$$

Labelled identifier ($\langle lvid, \overline{l}\rangle \downarrow lvid'$)

$$\frac{l \in \overline{l}}{\langle vid^{l}, \overline{l}\rangle \downarrow vid^{l}} \qquad \frac{l \notin \overline{l}}{\langle vid^{l}, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

Labelled labelled identifier ($\langle llvid, \overline{l}\rangle \downarrow llvid'$)

$$\frac{\langle lvid, \overline{l}\rangle \downarrow lvid' \quad l \in \overline{l}}{\langle lvid^{l}, \overline{l}\rangle \downarrow lvid'^{l}} \qquad \frac{\langle lvid, \overline{l}\rangle \downarrow lvid' \quad l \notin \overline{l}}{\langle vid^{l}, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lvid'\rangle))}$$

Constructor binding ($\langle conbind, \overline{l}\rangle \downarrow conbind'$)

$$\frac{l \in \overline{l}}{\langle vid^{l}, \overline{l}\rangle \downarrow vid^{l}} \qquad \frac{l \notin \overline{l}}{\langle vid^{l}, \overline{l}\rangle \downarrow \text{dots}(\varnothing)}$$

$$\frac{\langle lty, \overline{l}\rangle \downarrow lty' \quad \langle llvid, \overline{l}\rangle \downarrow llvid' \quad l \in \overline{l}}{\langle llvid \text{ of}^{l}\ lty, \overline{l}\rangle \downarrow llvid' \text{ of}^{l}\ lty'} \qquad \frac{\langle lty, \overline{l}\rangle \downarrow lty' \quad \langle llvid, \overline{l}\rangle \downarrow llvid' \quad l \notin \overline{l}}{\langle llvid \text{ of}^{l}\ lty, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle llvid', lty'\rangle))}$$

Constructor binding sequence ($\langle conbindseq, \overline{l}\rangle \downarrow conbindseq'$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle conbind_i, \overline{l}\rangle \downarrow conbind_i' \quad \text{seqFlatten}(\langle conbind_1', \ldots, conbind_n'\rangle) = \langle \text{dots}(\overrightarrow{part})\rangle}{\langle conbind_1 \mid \cdots \mid conbind_n, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle conbind_i, \overline{l}\rangle \downarrow conbind_i' \quad \text{seqFlatten}(\langle conbind_1', \ldots, conbind_n'\rangle) = \langle conbind_1'', \ldots, conbind_m''\rangle}{\langle conbind_1 \mid \cdots \mid conbind_n, \overline{l}\rangle \downarrow conbind_1'' \mid \cdots \mid conbind_m''}$$

Value binding ($\langle valbind, \overline{l}\rangle \downarrow valbind'$)

$$\frac{\langle lpat, \overline{l}\rangle \downarrow lpat' \quad \langle lexp, \overline{l}\rangle \downarrow lexp' \quad (l \in \overline{l} \text{ or } lpat' \neq \text{dots}(\varnothing))}{\langle \text{rec } lpat \stackrel{l}{=} lexp, \overline{l}\rangle \downarrow \text{rec } lpat' \stackrel{l}{=} lexp'} \qquad \frac{\langle lpat, \overline{l}\rangle \downarrow \text{dots}(\varnothing) \quad \langle lexp, \overline{l}\rangle \downarrow lexp' \quad l \notin \overline{l}}{\langle \text{rec } lpat \stackrel{l}{=} lexp, \overline{l}\rangle \downarrow \text{dots}(\text{flatten}(\langle lexp'\rangle))}$$

Datatype name ($\langle\langle tvseq, tc^l\rangle, \overline{l}\rangle \downarrow datname$)

$$\frac{\langle tvseq, \overline{l}\rangle \downarrow tvseq' \quad l \in \overline{l}}{\langle\langle tvseq, tc^l\rangle, \overline{l}\rangle \downarrow tvseq'\ tc^l}$$

$$\frac{\langle tvseq, \overline{l}\rangle \downarrow \text{dots}(\varnothing) \quad l \notin \overline{l}}{\langle\langle tvseq, tc^l\rangle, \overline{l}\rangle \downarrow \text{dots}(\varnothing)} \qquad \frac{\langle tvseq, \overline{l}\rangle \downarrow tvseq' \quad tvseq' \neq \text{dots}(\varnothing) \quad l \notin \overline{l}}{\langle\langle tvseq, tc^l\rangle, \overline{l}\rangle \downarrow tvseq'\ \text{dots}(\varnothing)}$$

Datatype binding ($\langle datbind, \overline{l}\rangle \downarrow datbind'$)

$$\frac{\langle\langle tvseq, tc^{l_1}\rangle, \overline{l}\rangle \downarrow datname \quad \langle conbindseq, \overline{l}\rangle \downarrow conbindseq' \quad (l_2 \in \overline{l} \text{ or } datname \neq \text{dots}(\varnothing) \text{ or } conbindseq' \neq \text{dots}(\overrightarrow{part}))}{\langle tvseq\ tc^{l_1} \stackrel{l_2}{=} conbindseq, \overline{l}\rangle \downarrow datname \stackrel{l_2}{=} conbindseq'}$$

$$\frac{\langle\langle tvseq, tc^{l_1}\rangle, \overline{l}\rangle \downarrow \text{dots}(\varnothing) \quad \langle conbindseq, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part}) \quad l_2 \notin \overline{l}}{\langle tvseq\ tc^{l_1} \stackrel{l_2}{=} conbindseq, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}$$

Datatype binding sequence ($\langle datbindseq, \overline{l}\rangle \downarrow datbindseq'$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle datbind_i, \overline{l}\rangle \downarrow datbind_i' \quad \text{seqFlatten}(\langle datbind_1', \ldots, datbind_n'\rangle) = \langle \text{dots}(\overrightarrow{part})\rangle}{\langle datbind_1 \text{ and } \cdots \text{ and } datbind_n, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ \langle datbind_i, \overline{l}\rangle \downarrow datbind_i' \quad \text{seqFlatten}(\langle datbind_1', \ldots, datbind_n'\rangle) = \langle datbind_1'', \ldots, datbind_m''\rangle}{\langle datbind_1 \text{ and } \cdots \text{ and } datbind_n, \overline{l}\rangle \downarrow datbind_1'' \text{ and } \cdots \text{ and } datbind_m''}$$

Declaration ($\langle dec, \overline{l}\rangle \downarrow dec$)

$$\frac{\langle valbind, \overline{l}\rangle \downarrow valbind' \quad valbind' \neq \text{dots}(\overrightarrow{part})}{\langle \text{val } valbind, \overline{l}\rangle \downarrow \text{val } valbind'} \qquad \frac{\langle valbind, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}{\langle \text{val } valbind, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}$$

$$\frac{\langle datbindseq, \overline{l}\rangle \downarrow datbindseq' \quad datbindseq' \neq \text{dots}(\overrightarrow{part})}{\langle \text{datatype } datbindseq, \overline{l}\rangle \downarrow \text{datatype } datbindseq'} \qquad \frac{\langle datbindseq, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}{\langle \text{datatype } datbindseq, \overline{l}\rangle \downarrow \text{dots}(\overrightarrow{part})}$$

Figure 14. Extension of the type system with slices

Labelled term ($lexp : \langle tse, \sigma\rangle$) ($lty : \langle tse, \sigma\rangle$) ($lpat : \langle tse, \sigma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Part ($part : tse$)

$$\frac{part \in \{exp, pat, ty\} \quad part : \langle tse, \sigma\rangle}{part : tse} \qquad \frac{dec : \langle tse, tse'\rangle}{dec : tse} \qquad \frac{typeseq : \langle tse, \overrightarrow{\sigma}\rangle}{typeseq : tse}$$

Type variable ($ltv : tv$)

$$\frac{}{tv^l : tv} \qquad \frac{}{\mathrm{dots}(\varnothing) : tv}$$

Type name ($ltc : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \nabla\rangle\rangle$)

$$\frac{tse(tc) = \overrightarrow{\alpha}\ \gamma}{tc^l : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \{tc \mapsto \overrightarrow{\alpha}\ \gamma\}\rangle\rangle} \qquad \frac{}{\mathrm{dots}(\varnothing) : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \varnothing\rangle\rangle}$$

Type variable sequence ($tvseq : \overrightarrow{tv}$)

$$\frac{ltv : tv}{ltv^l : \langle tv\rangle} \qquad \frac{}{\epsilon^l : \langle\rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ ltv_i : tv_i \quad n \geq 1}{(ltv_1, \ldots, ltv_n)^l : \langle tv_1, \ldots, tv_n\rangle} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ ltv_i : tv_i}{\mathrm{dots}(\langle ltv_1, \ldots, ltv_n\rangle) : \overrightarrow{tv}}$$

Type ($ty : \langle tse, \sigma\rangle$)

$$\frac{typeseq : \langle tse, \overrightarrow{\sigma}\rangle \quad ltc : \langle tse, \overrightarrow{\alpha}\ \gamma\rangle \quad |\overrightarrow{\alpha}| = |\overrightarrow{\sigma}|}{\lceil typeseq\ ltc\rceil^l : \langle tse, \overrightarrow{\sigma}\ \gamma\rangle} \qquad \frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Type sequence ($typeseq : \langle tse, \overrightarrow{\sigma}\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Constructor binding ($conbind : \langle tse, \langle \Gamma, \overrightarrow{\alpha}\ \gamma\rangle\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \langle \varnothing, \overrightarrow{\alpha}\ \gamma\rangle\rangle}$$

Constructor binding sequence ($conbindseq : \langle tse, \langle \Gamma, \overrightarrow{\alpha}\ \gamma\rangle\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \langle \Gamma, \overrightarrow{\alpha}\ \gamma\rangle\rangle}$$

Datatype binding sequence ($datbindseq : \langle tse, tse'\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \langle \varnothing, \varnothing\rangle\rangle}$$

Value binding ($valbind : \langle tse, \Gamma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \varnothing\rangle}$$

Datatype name ($datname : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \nabla\rangle\rangle$)

$$\frac{tvseq : \overrightarrow{\alpha} \quad ltc : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \nabla\rangle\rangle}{tvseq\ ltc : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \nabla\rangle\rangle} \qquad \frac{}{\mathrm{dots}(\varnothing) : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \varnothing\rangle\rangle}$$

Datatype binding ($datbind : \langle tse, \langle \nabla, \Gamma, \gamma\rangle\rangle$)

$$\frac{datname : \langle tse, \langle \overrightarrow{\alpha}\ \gamma, \nabla\rangle\rangle \quad conbindseq : \langle tse, \langle \Gamma, \overrightarrow{\alpha}\ \gamma\rangle\rangle}{datname \stackrel{l}{=} conbindseq : \langle tse, \langle \nabla, \Gamma, \gamma\rangle\rangle}$$

where $\mathrm{tyVarSet}(conbindseq) \subseteq \mathrm{ran}(\overrightarrow{\alpha})$

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \langle \varnothing, \varnothing, \gamma\rangle\rangle}$$

Declaration ($dec : \langle tse, tse'\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \langle \varnothing, \varnothing\rangle\rangle}$$

Value constructor ($lvid : \langle tse, \sigma\rangle$)

$$\frac{tse(vid) = \langle \overrightarrow{\sigma} \cup \{\sigma\}, \mathrm{c}\rangle}{vid^l : \langle tse, \sigma\rangle} \qquad \frac{}{\mathrm{dots}(\varnothing) : \langle tse, \sigma\rangle}$$

Atomic expression ($atexp : \langle tse, \sigma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Expression ($exp : \langle tse, \sigma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Match ($match : \langle tse, \sigma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Match rule ($mrule : \langle tse, \sigma\rangle$)

$$\frac{\forall part \in \mathrm{ran}(\overrightarrow{part}).\ part : tse}{\mathrm{dots}(\overrightarrow{part}) : \langle tse, \sigma\rangle}$$

Atomic pattern ($atpat : \langle tse, \sigma\rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i : \langle tse, \sigma_i\rangle}{\mathrm{dots}(\langle part_1, \ldots, part_n\rangle) : \langle tse, \sigma\rangle}$$

Pattern ($pat : \langle tse, \sigma\rangle$)

$$\frac{lvid : \langle tse, \sigma' \to \sigma\rangle \quad atpat : \langle tse, \sigma'\rangle}{\lceil lvid\ atpat\rceil^l : \langle tse, \sigma\rangle} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ part_i : \langle tse, \sigma_i\rangle}{\mathrm{dots}(\langle part_1, \ldots, part_n\rangle) : \langle tse, \sigma\rangle}$$

Figure 15. Extension of the constraint generator with slices

(In this figure only let $\overrightarrow{part} = \langle part_1, \ldots, part_n \rangle$ and $\overline{env} = \{env_1, \ldots, env_n\}$.)

Labelled term ($lexp \Downarrow \langle env, \alpha, \overline{c} \rangle$) ($lty \Downarrow \langle env, \alpha, \overline{c} \rangle$) ($lpat \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Part ($part \Downarrow \langle env, \overline{c} \rangle$)

$$\frac{part \in \{exp, pat, ty\} \quad part \Downarrow \langle env, \alpha, \overline{c} \rangle}{part \Downarrow \langle env, \overline{c} \rangle}$$

$$\frac{dec \Downarrow \langle env, env', \overline{c} \rangle}{dec \Downarrow \langle env, \overline{c} \rangle} \qquad \frac{typeseq \Downarrow \langle env, \langle \beta, \beta' \rangle, \overline{c} \rangle}{ty \Downarrow \langle env, \overline{c} \rangle}$$

Type variable ($ltv \Downarrow \langle ren, \alpha, \overline{c} \rangle$)

$$\frac{}{\mathrm{dots}(\varnothing) \Downarrow \langle \varnothing, \alpha, \varnothing \rangle}$$

Type variable sequence ($tvseq \Downarrow \langle ren, \langle \beta, \beta' \rangle, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ ltv_i \Downarrow \langle ren_i, \alpha_i, \overline{c}_i \rangle}{\mathrm{dots}(\langle ltv_1, \ldots, ltv_n \rangle) \Downarrow \langle ren, \langle \beta, \beta' \rangle, \overline{c} \rangle}$$

where $ren = ren_1 + \ldots + ren_n$
and $\mathrm{djv}(\mathrm{dom}(ren), \mathrm{ran}(ren_1), \ldots, \mathrm{ran}(ren_n), \alpha_1, \ldots, \alpha_n, \beta, \beta')$
and $\overline{c} = \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \cup \mathrm{mulConsT}(\langle ltv_1, \ldots, ltv_n \rangle)$

Labelled type constructor ($ltc \Downarrow \langle env, \langle \alpha, \beta, \beta' \rangle, \overline{c} \rangle$)

$$\frac{\mathrm{djv}(\alpha, \beta, \beta')}{\mathrm{dots}(\varnothing) \Downarrow \langle \mathrm{emEnv}, \langle \alpha, \beta, \beta' \rangle, \varnothing \rangle}$$

Value constructor ($lvid \Downarrow \langle env, \langle \alpha, \alpha', c \rangle, \overline{c} \rangle$)

$$\frac{}{\mathrm{dots}(\varnothing) \Downarrow \langle \mathrm{emEnv}, \langle \alpha, \alpha', \mathrm{con}(\varnothing, \varnothing) \rangle, \overline{c} \rangle}$$

Type ($ty \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Type sequence ($typeseq \Downarrow \langle env, \langle \beta, \beta' \rangle, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \beta, \beta')}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \langle \beta, \beta' \rangle, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Constructor binding ($conbind \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Constructor binding sequence ($conbindseq \Downarrow \langle env, \overline{\alpha}, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \overline{\alpha}, \overline{c} \rangle}$$

where $\forall i \in \{1, \ldots, n\}.\ \overline{var}_i = \mathrm{varSet}(\overline{c}_i) \setminus \mathrm{varSet}(part_i)$
and $\mathrm{dj}(\overline{var}_1, \ldots, \overline{var}_n, \cup_{i \in \{1,\ldots,n\}} \mathrm{varSet}(part_i), \overline{\alpha})$
and $\overline{c} = \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \cup \mathrm{mulConsI}(\uplus_{i \in \{1,\ldots,n\}} env_i|_{\mathrm{C}}, \mathrm{false})$

Value binding ($valbind \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Datatype name ($datname \Downarrow \langle \langle ren, env \rangle, \langle \alpha, \beta_1, \beta_1', \beta_2, \beta_2' \rangle, \overline{c} \rangle$)

$$\frac{\mathrm{djv}(\alpha, \beta_1, \beta_1', \beta_2, \beta_2')}{\mathrm{dots}(\varnothing) \Downarrow \langle \langle \varnothing, \varnothing \rangle, \langle \alpha, \beta_1, \beta_1', \beta_2, \beta_2' \rangle, \varnothing \rangle}$$

Datatype binding ($datbind \Downarrow \langle env, \langle tenv, cenv \rangle, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \langle \varnothing, \varnothing \rangle, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Datatype binding sequence ($datbindseq \Downarrow \langle env, \langle tenv, cenv \rangle, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \langle \varnothing, \varnothing \rangle, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Declaration ($dec \Downarrow \langle env, env', \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \mathrm{emEnv}, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Atomic expression ($atexp \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Expression ($exp \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Match ($match \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Match rule ($mrule \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Atomic pattern ($atpat \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$

Pattern ($pat \Downarrow \langle env, \alpha, \overline{c} \rangle$)

$$\frac{\forall i \in \{1, \ldots, n\}.\ part_i \Downarrow \langle env_i, \alpha_i, \overline{c}_i \rangle \quad \mathrm{djv}(\overline{c}_1, \ldots, \overline{c}_n, \alpha)}{\mathrm{dots}(\overrightarrow{part}) \Downarrow \langle \mathrm{uenv}(\overline{env}), \alpha, \cup_{i \in \{1,\ldots,n\}} \overline{c}_i \rangle}$$