UNIVERSITÀ DEGLI STUDI DI GENOVA
Scuola di scienze matematiche, fisiche e naturali
Dipartimento di informatica, bioingegneria, robotica e ingegneria dei sistemi
Corso di laurea magistrale in informatica

A set-theoretic type system for polymorphic variants in ML

Candidato Tommaso Petrucciani

Relatori Prof. Giuseppe Castagna
Prof. Elena Zucca
Prof. Davide Ancona

Correlatore Prof. Eugenio Moggi

Anno accademico 2014–2015

Contents

1 Introduction

This thesis investigates the application of the theory of set-theoretic types and semantic subtyping to the design of a type system for *polymorphic variants* – a feature of the OCaml programming language, studied in general for languages of the ML family. We develop a type system employing set-theoretic types, and we give new results on type reconstruction for such systems. We argue the system is well-suited to typing polymorphic variants and compares favourably to the one currently used in OCaml.

Type systems and semantic subtyping

Static type systems are powerful tools to help ensure the correctness of programs with respect to their specification. They consist in a system of rules assigning types to constructs in a program; these types are in a sense abstractions of the concrete data manipulated by the program at run-time (intuitively, we can see types as sets of concrete values).

In a *sound* static type system, the rules ensure that a well-typed program – one we can assign a type to – will satisfy properties that guarantee some kinds of errors will not occur. The downside is that a sound system necessarily rejects some programs which are type-safe – that is, which will not cause errors in practice – together with all the unsafe ones. An effective type system allows the programmer to encode many correctness properties into types, making well-typed programs provide stronger guarantees. It must also be flexible enough, lest programming become overly cumbersome.

This flexibility partly depends on *polymorphism* – the possibility of writing code that can work on data of different types – which allows us to write reusable code. There exist three main forms of polymorphism:

- *parametric polymorphism*, which allows us to write code generically and have it work on every type uniformly;
- *ad-hoc polymorphism*, which allows us to write code that can work on a few different types, possibly acting differently on each (*e.g.* overloading in many languages);

The work reported in this thesis has been done during a six-month traineeship at Laboratoire Preuves, Programmes et Systèmes of Université Paris Diderot, under the supervision of Giuseppe Castagna and Kim Nguyễn. The stay has been funded by an Erasmus+ grant.

- *subtyping*, where expressions may have many types organized in a hierarchy of more specific and more general types.

Another factor which can make programming in statically-typed languages cumbersome is that the programmer must often write annotations to guide the type-checker. This can be avoided if a language features *type inference* or *type reconstruction*: the capability of inferring type information that is not explicitly specified. However, increasing the sophistication of a type system may make reconstruction undecidable and hence make annotations necessary, at least to some extent.

SEMANTIC SUBTYPING In type systems with subtyping, there exists a subtyping relation between types that organizes them in hierarchy. We say this relation is *semantic* when it is defined from a semantic model, as opposed to being defined axiomatically by inductive or co-inductive rules: a semantic subtyping relation is derived by interpreting types as sets and seeing subtyping as set containment (Frisch, Castagna, and Benzaken, 2008). Such a semantic definition yields subtyping with an intuitive behaviour in the presence of set-theoretic type connectives, such as union and intersection types, when it can be difficult for axiomatic rules to be expressive enough.

Systems with semantic subtyping have first been studied in the context of typing transformations of XML documents, where they have been applied in the XDuce language (Hosoya and Pierce, 2003). We base our system on later work which extends semantic subtyping to systems with higher-order functions and parametric polymorphism (Castagna and Xu, 2011). The ℂDuce programming language implements this system (Benzaken, Castagna, and Frisch, 2003); it is particularly optimized for XML processing, but is in fact a general-purpose functional language.

Such systems have proven to be extremely flexible and expressive, as required by their original goal of typing semi-structured data precisely. They have also been applied to the design of type systems for NoSQL query languages (Benzaken, Castagna, Nguyễn, and Siméon, 2013).

Motivations

Polymorphic variants are a useful feature of OCaml that balances static safety and opportunities for code reuse with a remarkable conciseness. They offer a more flexible alternative to the ordinary variant types (or algebraic sum data types) of languages of the ML family. Essentially, polymorphic variants introduce a sort of subtyping whereby different variant types may share some cases.

In OCaml, they are typed with a form of parametric polymorphism called *structural polymorphism* (Garrigue, 2002), which superimposes a system of kinding constraints over the Hindley-Milner type system. This is used, in a

sense, to simulate subtyping without actually introducing it in the system. In general, the current system reuses the ML type system – including unification for type reconstruction – as much as possible.

We argue that using a different system, one that departs more significantly from ML, can be advantageous. In the current system, while constraints are somewhat separated from the rest (which is why unification and reconstruction can be extended to them), they still introduce significant complexity. Furthermore, structural polymorphism lacks some of the flexibility offered by a system with true subtyping, and it can result in unintuitive behaviour.

Contributions

The main contribution of this thesis is the definition of a type system for a fragment of ML including polymorphic variants. Our system employs a rich language of types, including set-theoretic type connectives, and features a subtyping relation which is defined semantically.

The system is based on earlier work for the polymorphic ℂDuce language, with some changes and restrictions. Notably, it does not allow typing overloaded functions with intersection types, and hence it removes ad-hoc polymorphism (we reintroduce it in an extension). We also disallow type-cases on functional types, a feature of ℂDuce which is unnecessary in our setting and, in general, of significant complexity and dubious utility in practice.

In our opinion, this system compares favourably to that used in OCaml. Firstly, it is more expressive: not only can it type every program the OCaml type system can type, but it can also type some useful type-safe programs that OCaml fails to type. Secondly, it is arguably more intuitive: we can express much more information at the level of types, which means we can do without the system of kinding constraints. This is made possible especially by the presence of set-theoretic type connectives: they allow us to encode bounded quantification – which is introduced in OCaml by structural polymorphism – without having to add it to the system.

Our system also models pattern matching precisely and quite intuitively. We can describe exhaustiveness and non-redundancy checking as subtype checking, whereas in OCaml they cannot be defined at the level of types.

We first give a deductive, non-syntax-directed presentation of the system; we can derive a typing algorithm if we assume functions to be annotated with their types. We then study type reconstruction with the aim of avoiding the need for these annotations. We build on earlier work by Castagna, Nguyễn, Xu, and Abate (2015), defining sound and complete reconstruction for a restriction of the system without let-polymorphism. This is possible because we have restricted the use of intersection types, as reconstruction would be undecidable if we admitted their use to type functions. Afterwards, we extend reconstruction to the full system with let-polymorphism and prove its soundness (but not its completeness). Our work on type reconstruction

is motivated by the practical application of this thesis, but also contributes to the study of reconstruction for set-theoretic type systems in general.

We discuss three variations on our system at the end of this thesis. Among them, we describe a refinement of the typing of pattern matching which is also applicable to ℂDuce and has been included in its development version.

Outline

Chapter 2 introduces the ML language family and OCaml in particular. It describes polymorphic variants as they are implemented in OCaml, including their issues and limitations; it presents the motivating examples of our work.

In Chapters 3 and 4, we describe two type systems for polymorphic variants. Chapter 3 presents the syntax, semantics, and type system that we take as our model of OCaml. It is adapted and extended somewhat with respect to published material; in particular, we formalize full pattern matching for concreteness, while other treatments use a restricted form.

Chapter 4 presents the system with set-theoretic types. We compare it with the system of Chapter 3 to show it can type any program that system can type. In this chapter, we give a deductive presentation of the system.

In Chapter 5, we study reconstruction for the set-theoretic type system of the previous chapter. We first remove let-polymorphism and define a sound and complete type reconstruction system for this simpler setting; then, we extend the system to allow let-polymorphism and prove its soundness.

In Chapter 6, we discuss three extensions and variations of the system: the addition of overloaded function, a refinement in the typing of pattern matching, and a restriction which solves a discrepancy between our model and OCaml (the lack of type tagging at run-time in the implementation).

Finally, in Chapter 7 we conclude by summarizing our work and pointing out some directions for future research.

2 *Polymorphic variants in OCaml*

This chapter introduces *polymorphic variants*, the programming language feature we will study in the rest of this work. In particular, we consider their implementation in the OCaml language. We begin by introducing the language and the ML family in general; then, we describe the purpose of polymorphic variants and present their semantics and typing in OCaml informally. In the next chapter, we will formalize these notions, to compare them later with our work.

At the end of this chapter, we list some aspects of the current system which can arguably be thought unintuitive or restrictive, and we provide the motivating examples of our work: functions that are type-safe but are ill-typed in OCaml or are given overly restrictive types.

2.1 THE ML LANGUAGE FAMILY AND OCAML

ML was originally the name of a programming language designed in the 1970s at the University of Edinburgh (Gordon, Milner, and Wadsworth, 1979). It has proved extremely influential, and its name is often used to refer to a family of languages related to it – among which OCaml – and to the type system they employ. ML uses call-by-value evaluation; it includes first-class functions and other features of functional languages, but also allows side-effects and mutability. It is statically typed and features type inference.

TYPE SYSTEM ML employs the Hindley-Milner type system (Damas and Milner, 1982), which extends the simply-typed λ-calculus with a form of parametric polymorphism called *let-polymorphism*. This is restricted in power with respect to the polymorphism of System F (Girard, 1972). As a consequence, type reconstruction for ML is decidable and can be made efficient in practice; hence, programmers need not write types explicitly.

Let-polymorphism distinguishes types from polymorphic *type schemes*, which represent families of types of a given form. For instance, the scheme $\forall\alpha.\,\alpha \to \alpha$ represents all types of the form $t \to t$, like $\mathsf{int} \to \mathsf{int}$; these are exactly all the types we can assign to the identity function $\lambda x.x$ ($\forall\alpha.\,\alpha \to \alpha$ is its *principal type scheme*).

Variables bound by λ-abstractions in a program, like function arguments, can only be assigned types and not type schemes. This means that the ar-

gument of a function cannot be used with multiple different types in the function body. Conversely, variables introduced by let declarations can be given type schemes. For example, the application

$$(\lambda i.\ (i\ \mathsf{true}, i\ 1))\ (\lambda x.x)$$

which would yield $(\mathsf{true}, 1)$, is ill-typed; we can rewrite it as

$$\mathsf{let}\ i = \lambda x.x\ \mathsf{in}\ (i\ \mathsf{true}, i\ 1)$$

which is well-typed: i is introduced by a let, so it can be given the type scheme $\forall \alpha.\ \alpha \to \alpha$ and thus be used with both types bool → bool and int → int.

OCAML OCaml (or Objective Caml; Leroy, 2014) is a dialect of the ML language which adds many additional features. These include objects and classes, a sophisticated module system including first-class modules, polymorphic variants, generalized algebraic data types (GADTs), and many others.

These features complicate the type system significantly. However, OCaml still features extensive type inference and minimizes the need for annotations.

Variants and pattern matching

Variant types – elsewhere called *algebraic sum data types* or *discriminated unions* – are a useful feature of ML-derived languages. We use them to define various data structures, from simple enumerations to inductively-defined data such as lists or trees.

In general, each value of a given variant type falls into one of a finite number of different cases, identified by *tags* or *labels*, and may carry additional information beside the tag itself. We must declare variant types explicitly by giving a set of tags and the type of argument associated to each tag. For instance, the type

```
type t = A of bool | B of int
```

is the discriminated union of bool and int: its values are booleans tagged by A (A true and A false) or integers tagged by B (*e.g.* B 2 or B 25).

Variants where tags have no argument type attached fulfil the role of the enumerated types of other languages. Those with arguments correspond to hierarchies of an interface and several concrete classes in Java-like languages.

Variant types can be recursive, so we can define inductive structures with them. They can also be polymorphic: we can use type variables in the arguments to define families of types. The following declaration defines binary trees where the leaves store values of type α.[1]

1 We do some pretty-printing of OCaml code throughout; for instance, we use Greek letters for type variables while OCaml uses 'a, 'b, 'c, ... We mark with ▸ lines showing types or error messages printed by the OCaml top-level interpreter.

```
type α tree = Leaf of α | Node of α tree * α tree
```

Lists in OCaml are defined analogously, though there is some syntactic sugar so we can write literal lists as `[]` or `[3; 2; 1]`.

We build expressions of variant types by writing a tag followed by an expression of the appropriate type for each argument: for example, `Int (1 + 3)` or `Node (Leaf 3, Leaf 4)`. We deconstruct them by *pattern matching*, which is a fundamental feature of languages of the ML family.

We use pattern matching to select different branches of execution depending on the shape of a value; we can also extract sub-terms of the value and bind them to variables. For instance, the recursive function

```
let rec sum tr = match tr with Leaf x → x | Node (tr1, tr2) → sum tr1 + sum tr2
▸ int tree → int
```

is defined on binary trees with integer leaves and computes the sum of all values in the leaves. Patterns can be nested to examine values in depth. Matching also subsumes the `let` construct: it allows us to extract polymorphic values from structures. For example, the following code is well-typed.

```
match ((fun x → x), 1) with (i, n) → (i n, i true)
```

OCaml issues a warning if a definition given by pattern matching does not cover all possible cases (an *exhaustiveness* check) or contains useless branches which will never be selected (a *non-redundancy* check). The function

```
let g n = match n with 0 → "zero" | 1 → "one" | 2 → "two" | 1 → "One"
```

triggers both warnings. It is defined only on three integers, but this cannot be expressed in its type (it is given type `int → string`), so the type system will assume it can be applied to any integer: the warning states it will raise an error if applied to, for instance, 3. Furthermore, the last branch is useless: patterns are checked in order, so the second occurrence of a repeated pattern is never selected.

Non-exhaustiveness warnings are crucial since they indicate situations that can lead to run-time exceptions (indeed, it is often advised to treat them as errors); they also aid in updating code to deal with changed data structures. Redundancy warnings also point out likely errors.

2.2 POLYMORPHIC VARIANTS

Polymorphic variants have been studied as a more flexible alternative to ordinary variants, giving greater possibilities for code reuse. They have been implemented first in the Objective Label compiler and then integrated into OCaml from version 3.0.

Garrigue (1998) describes polymorphic variants, though not in their very final implementation; some aspects concerning the reconstruction of pattern

matching are discussed in Garrigue (2004). Garrigue (2000) presents a detailed example of the possibilities of code reuse they offer.

Limitations of ordinary variants

Variant types must be declared before they are used by listing the set of cases, each with its tag and its argument type. This is in contrast to product types, for instance: a pair such as (1, 2) is given type int ∗ int, without having to declare this type.

While type declarations are useful for documentation, leaving them out is more concise and can still be clear as long as the type is only used in a small portion of the program. For example, two-value enumerations with meaningful names can be a more readable alternative to booleans (*e.g.* we can use Case_sensitive and Case_insensitive for a string comparison function). If we can omit declarations – as polymorphic variants allow – they can also be as concise.

A more significant limitation of variants is that different variant tags may not share cases. Consider the use of variant types to represent the grammar of expressions of a language. We might need different versions of the grammar, with more or fewer productions, to represent both the external language including syntactic sugar and internal ones manipulated during compilation. Presumably, these grammars will share productions.

Using a version of OCaml earlier than 4.01, the programmer would have to declare multiple types using different tag names, even for the common cases. Newer versions allow us to reuse the same tag. However, this is just a disambiguation performed by the compiler behind the scenes: different variant types are still distinct. For example, assume we have these two declarations.

```
type a = A of int | B of bool
type b = A of int | B of bool | C
```

The types share two tags: in a sense, we would say that a is a subtype of b, since every case of a is also a case of b. However, the two types are actually unrelated. A function like

```
let f = function A n → n >= 0 | B b → b | C → true
▸ b → bool
```

cannot be applied to an argument of type a, though it covers both the required cases.

Polymorphic variants allow a value to be used with multiple types which share that case (tag and argument type) but not necessarily other cases. This introduces a form of subtyping: for example, we can write code that works on some cases and reuse it for a type that only has a subset of them.

A related issue is that variant tags declared inside a module must be referred to by qualifying them with the module name. If the declaration of

type a above were in a module M, we would have to write M.A 2 instead of A 2, for instance. This is done to minimize conflicts caused by sharing tags; however it makes using variants declared in modules more cumbersome. With polymorphic variants, we need not qualify tags in this manner.

Syntax and semantics

The syntax and semantics of polymorphic variants are like those of ordinary ones. The only difference is that, since tags are not declared beforehand, we prefix them with a backquote to distinguish them from identifiers: for example, `A (1 + 3). We deconstruct polymorphic variant values by pattern matching, as for ordinary variants. For instance,

```
match `A (1 + 3) with `A x → x | `B → 0
```

yields 4. As for ordinary variants, polymorphic ones can also be without argument, like `B. Tuples can be used to write variants with multiple arguments, as in `C (3, 2).

A first look at the type system

Since we do not declare types, OCaml must generate them whenever we write a polymorphic variant expressions. The types it generates list the tags which may appear, together with their argument types.

```
`A 1
▸ [> `A of int ]
```

The > symbol means this type is compatible with types that have other tags. If we combine variant expressions with different tags, we get a type with multiple cases. Conversely, we may not combine in a type variants with the same tag unless the argument type is the same as well: this ensures the type of a variant argument can be predicted from its tag.

```
[ `A 1; `B true ]
▸ [> `A of int | `B of bool ] list
[ `A 1; `A true ]
▸ Error: This expression has type bool but an expression was expected of type int
```

Variables bound to variant values by let-declarations may also be assigned different types to combine them with different tags.

```
let x = `A 1 in (x, [ x; `B true ])
▸ [> `A of int ] * [> `A of int | `B of bool ] list
```

Functions defined by pattern matching have domain types with the < symbol, meaning they can be applied to arguments of a type with *fewer* cases

than those listed by pattern matching, not more. For instance,

```
  let f = function `A → true | `B → false
▸ [< `A | `B ] → bool
```

can be applied to variant types including at most the two cases `A and `B, both without arguments. We get types marked by > for functions defined on variants with any tag (*e.g.* when pattern matching includes a default case _).

```
  let g = function `A|`B → true | _ → false
▸ [> `A | `B ] → bool
```

If variant types with < are combined, we only keep the common cases. This occurs if we apply multiple functions to the same argument. For example, in

```
  let h1 = function `A|`B → true | `C → false
▸ h1 : [< `A | `B | `C ] → bool
  let h2 = function `A|`B → true | `D → false
▸ h2 : [< `A | `B | `D ] → bool
  let h3 x = (h1 x, h2 x)
▸ h3 : [< `A | `B ] → bool * bool
```

we define two functions which cover three tags each; h3 can only be applied to the two tags in common.

Delving deeper

The variant types we have shown in the examples above are actually type variables with constraints – called their *kinds* – attached. Variant expressions may be assigned multiple types by a mechanism called *structural polymorphism*, which is like the usual polymorphism of ML except for the addition of these constraints.

For example, the type of `A 1 is actually some variable $\alpha$, and [> `A of int] is the kind of α. OCaml omits the variable and just prints the kind unless the name is relevant (for instance, if the variable appears elsewhere in a type).

We have seen that `A 1 can be assigned different types – more precisely, type variables with different kinds – like [> `A of int] or [> `A of int | B of bool ]. This is because there is an *entailment* relation between kinds, and we can use any variable whose kind entails [> `A of int] to type `A 1.

There are also kinds with < constraints or with both forms. The latter are most often (but not exclusively) used 'internally' to type applications.

If we bind a variable to a variant expression using a let-declaration, we can then use the variable with any type whose kind entails the kind of the original expression. In

```
  let x = `A 1 in ([ x; `B true ], [ x; `B 2 ])
▸ [> `A of int | `B of bool ] list * [> `A of int | `B of int ] list
```

`A 1 is given the type '[> `A of int] as α'; then the type variable α is generalized and it is instantiated once to a variable of kind [> `A of int | `B of bool] and once to one of kind [> `A of int | `B of int].

Unlike types assigned to variables by let, those assigned to λ-abstracted variables cannot be generalized to polymorphic type schemes. Therefore, if we rewrite the code above without using let, it is rejected.

```
(fun x → ([ x; `B true ], [ x; `B 2 ])) (`A 1)
▸ Error: This expression has type int but an expression was expected of type bool
```

We conclude this overview by discussing two particular points.

BOUNDED QUANTIFICATION The typing of variants is a form of bounded quantification: when we quantify type variables we record their kinds, which then constrain how the variables can be instantiated.

When a type variable appears in both the domain and the codomain of a function type, the kind on its argument is thus propagated to its result, giving us precise static information. For example, consider an identity function on `A and `B defined as follows.

```
let id x = match x with `A|`B → x
▸ ([< `A | `B ] as α) → α
```

The kind [< `A | `B] bounds the types we can apply id to; intuitively we have the bounded-polymorphic type scheme $\forall \alpha \leq [\texttt{`A} \mid \texttt{`B}]. \alpha \to \alpha$. When we apply id, this kind is combined with the kind of the argument to determine that of the result. If we apply id to `A, we know statically that the result will itself be `A, which makes the following well-typed.

```
let f = function `A → true
▸ [< `A ] → bool
f (id `A)
▸ bool
```

On the contrary, an identity function defined without this sharing of variables does not allow the last application.

```
let id1 x = match x with `A → `A | `B → `B
▸ [< `A | `B ] → [> `A | `B ]
(function `A → true) (id1 `A)
▸ Error: This expression has type [> `A | `B ]
         but an expression was expected of type [< `A ]
         The second variant type does not allow tag(s) `B
```

The type of id1 corresponds intuitively to the bounded-polymorphic type scheme $\forall \alpha \leq [\texttt{`A} \mid \texttt{`B}], \beta \geq [\texttt{`A} \mid \texttt{`B}]. \alpha \to \beta$: fixing the instantiation of α does not give us any information on that of β.

However, the restrictions of let-polymorphism can make functions like id impractical, as we will see later.

CONJUNCTIVE TYPES Conjunctive types are a somewhat technical feature introduced to ensure that type inference reconstructs principal types (as discussed in Garrigue, 2002). When we combine kinds with < constraints and these kinds have a tag in common, but with different argument types, we get a conjunctive type.

```
  let f1 = function `A n → n + 1 | `B → 0
▸ [< `A of int | `B ] → int
  let f2 = function `A b → not b | `B → false
▸ [< `A of bool | `B ] → bool
  let f3 x = (f1 x, f2 x)
▸ [< `A of bool & int | `B ] → int * bool
```

The function f3 is given the arguably unintuitive type above, stating it can be applied to a `B or to an `A whose argument is both a boolean and an integer. Since this is impossible, a more intuitive type would be [< `B] → int * bool. The types behave alike in practice, but they are considered distinct.

2.3 SHORTCOMINGS OF OCAML AND PROPOSED EXTENSIONS

We point out a few limitations of the type system used in OCaml and a few areas in which our set-theoretic system will provide greater expressiveness.

LOSS OF POLYMORPHISM As we have said, variant polymorphism is handled via the instantiation of type variables. It is therefore limited by the restrictions ML imposes on parametric polymorphism in order to have complete type reconstruction: function arguments must be monomorphic.

In our type system, we use subtyping to describe variant polymorphism. We no longer lose polymorphism, in the sense that expressions of a variant type, even if they are function arguments, can always be combined with variants with tags different from their own. The following expressions will both be well-typed in our system, while only the former is in ML.

```
let x = `A 1 in ([ x; `B true ], [ x; `B 2 ])
(fun x → ([ x; `B true ], [ x; `B 2 ])) (`A 1)
```

This makes functions like id, above, more useful. In OCaml, as soon as we combine constraints we lose polymorphism. For example,

```
  [ id `A; `C ]
▸ Error: This expression has type [> `C ]
    but an expression was expected of type [< `A | `B > `A ]
    The second variant type does not allow tag(s) `C
```

is ill-typed, while [`A; `C] is well-typed (with type [> `A | `C] list).

We can make this well-typed with a *subtyping coercion*, but we must specify explicitly which tags we want to combine the result of id `A with.

```
[ (id `A :> [ `A | `C ]); `C ]
▸ [ `A | `C ] list
```

The result type is [`A | `C] list, which cannot be combined with other tags unless we use another coercion (it stands for [< `A | `C > `A | `C]). We cannot use coercions to obtain an open type like [> `A | `C].

We move to a system with subtyping, where subsumption can be applied to add new cases at any time (essentially making the coercion above implicit).

Another case in which OCaml produces functions whose results are insufficiently polymorphic is when we include a default case in pattern matching. For instance, the function

```
let id2 = function `A → `A | x → x
▸ ([> `A ] as α) → α
```

has the type above, meaning it can only be applied to variant expressions which admit the case `A. Its result will always have a type that includes `A; hence the application below is ill-typed, despite being type-safe.

```
(function `B → 2) (id2 `B)
▸ Error: This expression has type [> `A | `B ]
    but an expression was expected of type [< `B ]
    The second variant type does not allow tag(s) `A
```

The rationale behind this behaviour is explained in Garrigue (2004): it is simpler to describe in the kinding system, and it aids in detecting misspelled tags in pattern matching. However, it also rejects perfectly valid programs. We feel that removing this behaviour would be more intuitive; it might delay the detection of certain errors, but this risk can probably be mitigated by adding some type annotations.

TYPING OF PATTERN MATCHING Typing pattern matching requires us to determine the types for the capture variables of each pattern: we need them to make sense of the corresponding branch. OCaml considers each pattern separately, whereas our system takes preceding patterns into account as well to produce more precise types. Additionally, in our system we look for redundant branches and exclude them from type checking.

As an example of the first aspect, consider the following function (annotated with its type for clarity).

```
let f1: [< `A | `B ] → int = function `A → 1 | y → (function `B → 1) y
▸ Error: This expression has type [< `A | `B > `A ]
    but an expression was expected of type [< `B ]
    The second variant type does not allow tag(s) `A
```

OCaml considers the second branch ill-typed: it assumes y could be `A – because `A is in the domain – and concludes that the function function `B → 1 cannot be applied safely to y, being defined only on `B. However, y cannot

in fact be `A: if it were, the first pattern would have been selected and we would not have reached the second. Hence, it would be sound to consider the function well-typed. It is indeed well-typed in our system, because we take the preceding pattern into account to determine that y must be `B.

This is already possible to some extent in OCaml, but we must write it explicitly as

```
let f2: [< `A | `B ] → int = function `A → 1 | `B as y → (function `B → 1) y
▸ [< `A | `B ] → int
```

while in our system it is implicit and more general (it affects non-variant types as well).

We have said we exclude redundant branches, not typing them at all. Since such branches will never be selected, we do not need to type them to ensure they are safe. We also exclude them from the output type: a function will never yield results produced by a branch that is never selected. OCaml, conversely, considers all branches to determine the result type. We notice this in the following example.

```
let g1 = function `A → `A | `B → `B
▸ [< `A | `B ] → [> `A | `B ]
let g2: [< `A] → [> `A] = function `A → `A | `B → `B
▸ Warning 11: this match case is unused.
▸ [< `A ] → [> `A | `B ]
```

By adding a type annotation, we restrict our function so it only works on `A rather than on both `A and `B. This means the result will always be `A; however, this is not recognized by OCaml, which still considers both `A and `B to be possible results. Unused branches should not be considered in the output type, and a more precise type should be [< `A ] → [> `A].

RECONSTRUCTION AND EXHAUSTIVENESS In OCaml, type reconstruction for pattern matching expressions may have to determine the type of the matched expression from the patterns. For instance, in

```
type t = A of bool | B of int | C
fun x → match x with A x → x | B y → y >= 0
▸ Warning 8: this pattern–matching is not exhaustive.
    Here is an example of a value that is not matched: C
▸ t → bool
```

OCaml determines the type of the function from the two cases mentioned by pattern matching. However, this type makes matching not exhaustive, because C is not covered. In OCaml, non-exhaustive pattern-matching constructs are not rejected outright because they can be convenient sometimes (especially with ordinary variants).

With polymorphic variants, type reconstruction can generate new polymorphic variant types to make matching exhaustive. Instead of reusing some

type with three tags because two of them are mentioned, we use a new type with just the two tags that are actually mentioned. However, the interaction with other type constructors – pairs, in particular – complicates this. We might be able to make matching exhaustive only by restricting a polymorphic variant type so that some branches become useless. For instance, consider the following function (from Garrigue, 2004, which discusses this in detail).

```
let f = function (`A, _) → 1 | (`B, _) → 2 | (_, `A) → 3 | (_, `B) → 4
```

OCaml chooses the type [> `A | `B] * [> `A | `B] → int and makes matching not exhaustive. Choosing [< `A | `B] * [< `A | `B] → int makes it exhaustive, but it means the last two branches become redundant. Choosing a different solution, like [> `A | `B] * [< `A | `B] → int, would break symmetry, which is unintuitive.

Adding union types solves these problems, because we can avoid the approximation we have to make in OCaml for pair types. We would need the input type to be ([< `A | `B] * [>]) ∨ ([>] * [< `A | `B]), where ∨ denotes the union of two types and [>] a type which admits any polymorphic variant. This type would contain the values (`A, `C) and (`C, `A), but not (`C, `C). It is not expressible in OCaml, where reconstruction approximates it with the type [> `A | `B] * [> `A | `B], which also contains (`C, `C) – making matching non-exhaustive.

Thanks to unions and to singleton types – types containing a single constant – we can always find a type that makes matching exhaustive without being too restrictive and without making branches useless (when they would be useful with another choice of type). This means we can reconstruct 'intuitive' types without having to choose – possibly inconsistently in different cases – to favour either exhaustiveness or non-redundancy. We forbid non-exhaustive pattern matching altogether in our system, since this increased flexibility means it is never convenient.

3 *A calculus for ML with variants*

In this chapter, we introduce the *Variants* calculus, which we will consider from now on as our formalization of polymorphic variants in OCaml. It consists in a λ-calculus with constants, pairs, variants,[1] and pattern matching. We do not include recursive functions (though we may employ them in examples) as their addition seems to be orthogonal to that of the features we are considering. Likewise, we disregard imperative effects and mutability, though they exist in ML-derived languages like OCaml.

We first present the syntax and an operational semantics for the untyped calculus. We then describe *VariantsK*, a type system in which polymorphic variants are typed with type variables and a notion of *kind* is used to express constraints on these variables – as is done in OCaml and as we have explained informally in the previous chapter.

The design of this type system is motivated by the desire to have a minimal extension of ML and to keep the same language of types: hence the use of type variables – rather than types of some new form – for variant expressions. There is no direct notion of subtyping: while it may seem natural to ascribe variant polymorphism to subtyping, it is handled here through *structural polymorphism*, a form of parametric polymorphism. We thus retain the overall structure of the Hindley-Milner type system, despite the addition of constraints. This approach notably means that reconstruction can employ the standard unification algorithm used in ML, albeit extended to deal with kinds. Nevertheless, the differences from ML are significant, and they give rise to a much more complex system.

VariantsK is adapted from the type system presented in Garrigue (2002) and Garrigue (2015). We have omitted the abstract framework of constraint domains for the sake of concreteness and simplicity; we have also added full pattern matching for ease of comparison with the set-theoretic system. We only describe the deductive type system, not type reconstruction: we will tackle that problem directly in our set-theoretic type system.

The first two sections of this chapter present the syntax and semantics of the *Variants* calculus. The third describes *VariantsK*, and the fourth and last compares it to Garrigue's system, as well as to OCaml itself, to discuss the differences we have introduced in our model.

1 From now on, we use 'variants' rather than 'polymorphic variants' when referring to our calculus, and we distinguish the two only when dealing specifically with OCaml.

3.1 SYNTAX

We first define the syntax of expressions in *Variants*. Expressions include a pattern-matching construct; this depends on the syntax of patterns, which we present immediately afterwards.

We assume that there exists a countable set $\mathcal{X}$ of *expression variables*, ranged over by x, y, z, ... We also consider a set C of language constants, ranged over by c, and a set $\mathcal{L}$ of tags, ranged over by $`tag$. Tags are used to label variant expressions.

DEFINITION 3.1: Expressions An *expression* e is a term inductively generated by the following grammar:

$e ::= x$		variable
$\mid c$		constant
$\mid \lambda x.e$		abstraction
$\mid e\ e$		application
$\mid (e, e)$		pair
$\mid `tag(e)$		variant
$\mid$ match e with $(p_i \to e_i)_{i \in I}$		pattern matching

where p ranges over the set $\mathcal{P}$ of patterns, defined below. We write $\mathcal{E}$ to denote the set of all expressions.

As usual, we consider expressions up to α-renaming of the variables bound by abstractions and by patterns.

We define $fv(e)$ to be the set of expression variables occurring free in the expression e, and we say that e is *closed* if and only if $fv(e)$ is empty. □

Expressions include the three forms of the pure λ-calculus: variables, abstractions (where $\lambda x.e$ is the function of argument x and body e), and applications. We add constants and pairs; naturally, we include variant expressions as well. We only consider variants with a single argument since we can encode multiple arguments with pairs – as OCaml does in the case of polymorphic variants – and the absence of an argument with a dummy one, such as a constant () ('unit'). We include full pattern matching, with an arbitrary number of branches, with the match construct, where $I = \{1, \ldots, n\}$ for some positive n.

There is no let construct to bind identifiers. Its purpose in ML is to introduce polymorphic bindings by generalizing types into type schemes. In OCaml and other languages, pattern matching serves this role and therefore subsumes let. We follow the same approach and add let as syntactic sugar:

$$\text{let } x = e_0 \text{ in } e_1 \equiv \text{match } e_0 \text{ with } x \to e_1 .$$

DEFINITION 3.2: Patterns A *pattern* p is a term inductively generated by the following grammar:

$$
\begin{array}{lll}
p ::= & \_ & \text{wildcard} \\
\mid & x & \text{variable} \\
\mid & c & \text{constant} \\
\mid & (p, p) & \text{pair} \\
\mid & `tag(p) & \text{variant} \\
\mid & p \& p & \text{intersection} \\
\mid & p | p & \text{union}
\end{array}
$$

which satisfies the following constraints:

- in a pair pattern (p_1, p_2) or an intersection pattern $p_1 \& p_2$, the sets of expression variables appearing in p_1 and p_2 are disjoint;
- in a union pattern $p_1 | p_2$, the sets of expression variables appearing in p_1 and p_2 are equal.

We write $\mathcal{P}$ to denote the set of all patterns.

We write $capt(p)$ to denote the set of expression variables occurring as sub-terms in a pattern p, and we say they are the *capture variables* of p. □

A pattern plays two roles in pattern matching: accepting or refusing values and introducing bindings. Any value either matches the pattern or not; this determines whether a certain branch is followed or skipped. Additionally, when a pattern accepts a value and its corresponding branch is selected, it binds some sub-terms of that value to its capture variables; these sub-terms replace the capture variables wherever they occur in the branch.

Intuitively, the semantics of these patterns are as follows. Variable patterns and wildcards accept any value. Constants only accept themselves. Pair patterns only accept pairs and only if each sub-pattern accepts the corresponding component. Variant patterns only accept variants with the same tag, if the tag argument matches the inner pattern. Intersection patterns accept values accepted by both sub-patterns and unions those accepted by one at least (the first sub-pattern is tested first).

As for bindings, wildcards and constants do not bind anything, while a variable pattern x binds the accepted value to x. Pair, variant, and intersection patterns bind any variable bound by their sub-patterns; unions bind those bound by the successful one.

Intersection patterns such as we present here do not exist in OCaml. They are a generalization of *alias patterns* of the form p as x, where we do not force the second sub-pattern to be a variable.

$$\textit{R-Appl}\ \frac{}{(\lambda x.e)\ v \ \rightsquigarrow\ e[v/x]} \qquad \textit{R-Match}\ \frac{v/p_j = \varsigma \qquad \forall i < j.\ v/p_i = \Omega}{\mathsf{match}\ v\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \ \rightsquigarrow\ e_j\varsigma}\ j \in I$$

$$\textit{R-Ctx}\ \frac{e \ \rightsquigarrow\ e'}{E[\,e\,] \ \rightsquigarrow\ E[\,e'\,]}$$

FIGURE 3.1 Small-step reduction relation.

3.2 SEMANTICS

We define the operational semantics of *Variants* in small-step style, with a call-by-value evaluation strategy. We begin by giving standard definitions for values and evaluation contexts.

DEFINITION 3.3: Values A *value* v is a closed expression inductively generated by the following grammar.

$$v ::= c \mid \lambda x.e \mid (v, v) \mid \text{`}tag(v)$$ □

We use evaluation contexts to impose call-by-value evaluation in left-to-right order. An evaluation context, as defined below, is an expression with a hole placed in it so as to implement this order.

DEFINITION 3.4: Evaluation contexts Let the symbol [ ] denote a hole. An *evaluation context* E is a term inductively generated by the following grammar.

$$\begin{array}{lll} E ::= & [\,] & \\ \mid & E\ e & \mid\ v\ E \\ \mid & (E, e) & \mid\ (v, E) \\ \mid & \text{`}tag(E) & \mid\ \mathsf{match}\ E\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \end{array}$$

We write $E[\,e\,]$ for the expression obtained by replacing the hole in E with the expression e. □

We now define the reduction relation of our semantics, as well as the semantics of pattern matching on which reduction itself depends.

DEFINITION 3.5: Expression substitution An *expression substitution* ς is a partial mapping of expression variables to values. We write $[v_1/x_1, \ldots, v_n/x_n]$ for the substitution which replaces free occurrences of x_i with v_i, for each $i \in I$. We write $e\varsigma$ for the application of the substitution to an expression e. We write $\varsigma_1 \cup \varsigma_2$ for the union of disjoint substitutions. □

$$
\begin{aligned}
v/\_ &= [\,] \\
v/x &= [v/x] \\
v/c &= \begin{cases} [\,] & \text{if } v = c \\ \Omega & \text{otherwise} \end{cases} \\
v/(p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2),\ v_1/p_1 = \varsigma_1, \text{ and } v_2/p_2 = \varsigma_2 \\ \Omega & \text{otherwise} \end{cases} \\
v/\backprime tag(p_1) &= \begin{cases} \varsigma_1 & \text{if } v = \backprime tag(v_1) \text{ and } v_1/p_1 = \varsigma_1 \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v/p_1 = \varsigma_1 \text{ and } v/p_2 = \varsigma_2 \\ \Omega & \text{otherwise} \end{cases} \\
v/p_1 | p_2 &= \begin{cases} v/p_1 & \text{if } v/p_1 \neq \Omega \\ v/p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 3.2 Semantics of pattern matching.

DEFINITION 3.6: Reduction The reduction relation $\rightsquigarrow$ between expressions is given by the rules in Figure 3.1. □

DEFINITION 3.7: Semantics of pattern matching We write v/p for the result of matching a value v against a pattern p. We have either $v/p = \varsigma$, where ς is a substitution defined on the variables in $capt(p)$, or $v/p = \Omega$. In the former case, we say that v matches p (or that p accepts v); in the latter, we say that matching fails.

The definition of v/p is given in Figure 3.2. □

There are three rules: two notions of reduction and the rule for context closure which allows us to apply reduction to expressions in a context.

The rule *R-Appl* is the ordinary rule for call-by-value β-reduction. It states that the application of an abstraction $\lambda x.e$ to a value v reduces to the body e of the abstraction, where x is replaced by v.

R-Match depends on the semantics of pattern matching. It states that a match expression on a value v reduces to the branch e_j corresponding to the first pattern p_j for which matching is successful. The obtained substitution is applied to e_j, replacing the capture variables of p_j with sub-terms of v. If no pattern accepts v, the expression is stuck.

The matching operation implements the intuitive semantics we have just described. Patterns which bind no variable generate the empty substitution. Matching against pair or intersection patterns yields the union of two substitutions: note that the constraints we have imposed require the two to be

disjoint. Conversely, in a union pattern both branches must contain the same variables, so the domain of the resulting substitution is the same regardless of which pattern is selected.

3.3 TYPE SYSTEM

In this section, we describe the *VariantsK* type system for our calculus. It consists essentially in the core ML type system (the simply-typed λ-calculus augmented with let-polymorphism) with the addition of a notion of kinding to distinguish normal type variables from *constrained* ones.

Unlike the others, these constrained variables may not be instantiated into any type, but only into other variables with compatible constraints. They are used to type variant expressions: there are no 'variant types' *per se*. Constraints are recorded in *kinds* and kinds in a *kinding environment* which is included in the typing judgment.

We start by defining types for this system. We assume that there exists a countable set $\mathcal{V}$ of *type variables*, ranged over by $\alpha, \beta, \gamma, \ldots$. We also consider a finite set $\mathcal{B}$ of *basic types*, ranged over by b, and a function $b_{(\cdot)}$ from constants to basic types. For instance, we might take $\mathcal{B} = \{\mathsf{bool}, \mathsf{int}, \mathsf{unit}\}$, with $b_{\mathsf{true}} = \mathsf{bool}$, $b_{()} = \mathsf{unit}$, and so on.

DEFINITION 3.8: Types A *type* τ is a term inductively generated by the following grammar.

$$
\begin{array}{lll}
\tau ::= \alpha & & \text{type variable} \\
\quad \mid b & & \text{basic} \\
\quad \mid \tau_1 \to \tau_2 & & \text{arrow} \\
\quad \mid \tau_1 \times \tau_2 & & \text{product}
\end{array}
$$

□

We define kinds next. In a typing judgment, each type variable must be assigned a kind: the unconstrained kind for 'normal' variables and a constrained one for variables used to type polymorphic variants.

Kinds describe which tags may or may not appear (a *presence* information) and which argument types are associated to each tag (a *typing* information). The presence information is split in two parts, a lower and an upper bound. This is necessary to provide an equivalent to both covariant and contravariant subtyping – without actually having subtyping in the system – that is, to allow both variant values and functions defined on variant values to be polymorphic.

DEFINITION 3.9: Kinds A *kind* κ is either the *unconstrained kind* $\bullet$ or a *constrained kind*, that is, a triple (L, U, T) where:

- L is a finite set of tags $\{\text{`}tag_1, \ldots, \text{`}tag_n\}$;

$$
\begin{aligned}
[>\ \text{`A of int} \mid \text{`B of bool }]\ \text{as}\ \alpha &\equiv \alpha \ \text{ where } \ \alpha :: (\{\text{`A}, \text{`B}\}, \mathcal{L}, \{\text{`A: int}, \text{`B: bool}\}) \\
[<\ \text{`A of int} \mid \text{`B of bool }]\ \text{as}\ \beta &\equiv \beta \ \text{ where } \ \beta :: (\varnothing, \{\text{`A}, \text{`B}\}, \{\text{`A: int}, \text{`B: bool}\}) \\
[<\ \text{`A of int} \mid \text{`B of bool \& unit} >\ \text{`A }]\ \text{as}\ \gamma &\equiv \gamma \ \text{ where } \ \gamma :: (\{\text{`A}\}, \{\text{`A}, \text{`B}\}, \{\text{`A: int}, \text{`B: bool}, \text{`B: unit}\})
\end{aligned}
$$

FIGURE 3.3 Variant types and kinds in OCaml and *VariantsK*.

- U is either a finite set of tags or the set $\mathcal{L}$ of all tags;
- T is a finite set of pairs of a tag and a type, written $\{\text{`}tag_1\text{: } \tau_1, \ldots, \text{`}tag_n\text{: } \tau_n\}$ (its domain $dom(T)$ is the set of tags occurring in it);

and where the following conditions hold:

- $L \subseteq U$, $L \subseteq dom(T)$, and, if $U \neq \mathcal{L}$, $U \subseteq dom(T)$;
- tags in L have a single type in T, that is, if $\text{`}tag \in L$, whenever both $\text{`}tag\text{: } \tau_1 \in T$ and $\text{`}tag\text{: } \tau_2 \in T$, we have $\tau_1 = \tau_2$.

We define an entailment relation $\cdot \vDash \cdot$ between constrained kinds as

$$(L, U, T) \vDash (L', U', T') \iff L \supseteq L' \wedge U \subseteq U' \wedge T \supseteq T' \,. \qquad \square$$

Note that T may associate more than one type to any tag that is not in L: this corresponds to the conjunctive types we have seen in the previous chapter. T might also include types for tags that do not appear in U, though they are not very significant in general; Garrigue (2002) explains their relevance and the possibility of discarding them.

In OCaml, kinds are written with the typing information inlined in the lower and upper bounds. These are introduced by > and < respectively and, if missing, $\varnothing$ is assumed for the lower bound and $\mathcal{L}$ for the upper. Figure 3.3 shows three examples of kinds written in OCaml and in this formalism.

We next define kinding environments, which associate kinds to type variables. After them we introduce type schemes: schemes include a kinding environment to record the kinds of all variables they quantify.

DEFINITION 3.10: Kinding environments A *kinding environment* K is a partial mapping from type variables to kinds. We write kinding environments as $K = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$.

We write K, K' for the updating of the kinding environment K with the new bindings in K'. It is defined as follows.

$$(K, K')(\alpha) = \begin{cases} K'(\alpha) & \text{if } \alpha \in dom(K') \\ K(\alpha) & \text{otherwise} \end{cases}$$

We say that a kinding environment is *closed* if all the type variables that appear in the types in its range also appear in its domain. We say it is *canonical* if it is infinite and contains infinitely many variables of every kind. $\square$

Kinding environments effectively introduce recursive types into this system: while types are defined inductively, kinds may be recursive (possibly mutually so).

DEFINITION 3.11: Type schemes A *type scheme* σ is of the form $\forall A.\, K \triangleright \tau$, where:

- A is a finite set $\{\alpha_1, \ldots, \alpha_n\}$ of type variables;
- K is a kinding environment such that $\mathit{dom}(K) = A$.

We identify a type scheme $\forall \varnothing.\, \varnothing \triangleright \tau$, which quantifies no variable, with the type τ itself; thus, we view types as a subset of type schemes. Furthermore, we consider type schemes up to renaming of the variables they bind, and we disregard useless quantification (*i.e.* quantifying variables which do not occur in the type). □

We define the set of variables occurring in a type, and of variables occurring free in a type scheme, by taking into account a kinding environment. Thus, if a constrained variable α appears in a type, the free variables of this type will also include the variables that appear in the typing component of the kind of α.

DEFINITION 3.12: Free variables The set of *free variables* $\mathit{var}_K(\sigma)$ of a type scheme σ with respect to a kinding environment K is the minimum set satisfying the following equations.

$$
\begin{aligned}
\mathit{var}_K(\forall A.\, K' \triangleright \tau) &= \mathit{var}_{K,K'}(\tau) \setminus A \\
\mathit{var}_K(\alpha) &= \begin{cases} \{\alpha\} \cup \bigcup_{\mathit{tag}:\tau \in T} \mathit{var}_K(\tau) & \text{if } \alpha :: (L, U, T) \in K \\ \{\alpha\} & \text{otherwise} \end{cases} \\
\mathit{var}_K(b) &= b \\
\mathit{var}_K(\tau_1 \to \tau_2) &= \mathit{var}_K(\tau_1) \cup \mathit{var}_K(\tau_2) \\
\mathit{var}_K(\tau_1 \times \tau_2) &= \mathit{var}_K(\tau_1) \cup \mathit{var}_K(\tau_2)
\end{aligned}
$$

We say that a type τ is *ground* or *closed* if and only if $\mathit{var}_\varnothing(\tau)$ is empty. We say that a type or a type scheme is *closed in a kinding environment* K if all its free variables are in the domain of K. □

ML uses type substitutions to allow the instantiation of type schemes; a type substitution may replace the quantified variables of the scheme with any type. The addition of kinds changes this: variables with constrained kinds may only be instantiated into other variables with 'stronger' constraints (*i.e.* with a kind which entails the former one) and not with any other type. To express this, we add a notion of *admissibility* to the standard definition of type substitutions.

DEFINITION 3.13: Type substitutions A *type substitution* θ is a finite mapping of type variables to types. We write $[\,{}^{\tau_i}/_{\alpha_i} \mid i \in I\,]$ for the type substitution which simultaneously replaces α_i with τ_i, for each $i \in I$. We write $\tau\theta$ for the application of a substitution to a type τ, which is defined as follows.

$$\begin{aligned}
\alpha\theta &= \begin{cases} \tau' & \text{if } {}^{\tau'}/_{\alpha} \in \theta \\ \alpha & \text{otherwise} \end{cases} \\
b\theta &= b \\
(\tau_1 \to \tau_2)\theta &= (\tau_1\theta) \to (\tau_2\theta) \\
(\tau_1 \times \tau_2)\theta &= (\tau_1\theta) \times (\tau_2\theta)
\end{aligned}$$

We extend the $\mathit{var}_{(\cdot)}(\cdot)$ operation to substitutions as

$$\mathit{var}_K(\theta) = \bigcup_{\alpha \in \mathit{dom}(\theta)} \mathit{var}_K(\alpha\theta)\,.$$

We extend application of substitutions to the typing component of a constrained kind (L, U, T): $T\theta$ is given by the pointwise application of θ to all types in T. We extend it to kinding environments: $K\theta$ is given by the pointwise application of θ to the typing component of every kind in the range of K. We extend it to type schemes $\forall A.\, K \triangleright \tau$: by renaming quantified variables, we assume $A \cap (\mathit{dom}(\theta) \cup \mathit{var}_{\varnothing}(\theta)) = \varnothing$, and we have $(\forall A.\, K \triangleright \tau)\theta = \forall A.\, K\theta \triangleright \tau\theta$.

We write $\theta_1 \cup \theta_2$ for the union of disjoint substitutions and $\theta_2 \circ \theta_1$ for the composition of substitutions. □

DEFINITION 3.14: Admissibility of a type substitution We say that a substitution θ is *admissible* between two kinding environments K and K', and write $K \vdash \theta : K'$, when it preserves kinds:

$$\begin{aligned}
&\forall \alpha :: (L, U, T) \in K. \\
&\quad \big(\alpha\theta :: (L', U', T') \in K' \;\wedge\; (L', U', T') \vDash (L, U, T\theta)\big)
\end{aligned}$$

that is, whenever α is constrained in K, $\alpha\theta$ must be a type variable, it must be constrained in K', and its kind must entail the substitution instance of the kind of α in K. □

We now introduce type environments, which map expression variables to type schemes. Then, we define the operation of *generalization* with respect to a kinding environment and a type environment; generalization produces type schemes by quantifying all variables which do not appear free in the type environment. In the typing rules, we combine generalization with the typing rule for pattern-matching expressions rather than having a separate rule for it.

We also introduce *instantiation* of type schemes by defining the set of types that are instances of a given scheme.

DEFINITION 3.15: Type environments A *type environment* Γ is a partial mapping from expression variables to type schemes. We write type environments as $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$.

We write Γ, Γ' for the updating of the type environment Γ with the new bindings in Γ'. It is defined as follows.

$$(\Gamma, \Gamma')(x) = \begin{cases} \Gamma'(x) & \text{if } x \in dom(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$$

We extend the $var_{\cdot}(\cdot)$ operation to type environments as

$$var_K(\Gamma) = \bigcup_{\sigma \in range(\Gamma)} var_K(\sigma) \,. \qquad \square$$

DEFINITION 3.16: Generalization We define the *generalization* of a type τ with respect to a kinding environment K and a type environment Γ as the type scheme

$$gen_{K;\Gamma}(\tau) = \forall A.\, K' \triangleright \tau$$

where $A = var_K(\tau) \setminus var_K(\Gamma)$ and $K' = \{\, \alpha :: \kappa \in K \mid \alpha \in A \,\}$.

We extend this definition to type environments which only contain types (*i.e.* trivial type schemes) as

$$gen_{K;\Gamma}(\{x_1 : \tau_1, \ldots, x_n : \tau_n\}) = \{x_1 : gen_{K;\Gamma}(\tau_1), \ldots, x_n : gen_{K;\Gamma}(\tau_n)\} \,. \qquad \square$$

DEFINITION 3.17: Instances of a type scheme We define the *instances* of a type scheme $\forall A.\, K' \triangleright \tau$ in a kinding environment K as the set of types

$$inst_K(\forall A.\, K' \triangleright \tau) = \{\, \tau\theta \mid dom(\theta) \subseteq A \wedge K, K' \vdash \theta : K \,\} \,.$$

We say that a type scheme σ_1 is *more general* than a type scheme σ_2 in K, and we write $\sigma_1 \sqsubseteq_K \sigma_2$, if $inst_K(\sigma_1) \supseteq inst_K(\sigma_2)$.

We extend this notion to type environments as

$$\Gamma_1 \sqsubseteq_K \Gamma_2 \iff dom(\Gamma_1) = dom(\Gamma_2) \wedge \forall x \in dom(\Gamma_1).\, \Gamma_1(x) \sqsubseteq_K \Gamma_2(x) \,. \qquad \square$$

Note the condition of admissibility of the substitution, which is added with respect to a standard definition of instantiation in ML.

Finally, we define the typing relation itself.

DEFINITION 3.18: Typing relation The typing relation $K; \Gamma \vdash_{\kappa} e : \tau$ (*e is given type τ in the kinding environment K and the type environment Γ*) is defined by the rules in Figure 3.4, where we require K to be closed and Γ and τ to be closed with respect to K. We also assume that K is canonical. $\square$

$$\textit{Tk-Var}\ \frac{\tau \in \mathit{inst}_K(\Gamma(x))}{K;\Gamma \vdash_\kappa x\colon \tau} \qquad \textit{Tk-Const}\ \frac{}{K;\Gamma \vdash_\kappa c\colon b_c}$$

$$\textit{Tk-Abstr}\ \frac{K;\Gamma,\{x\colon \tau_1\} \vdash_\kappa e\colon \tau_2}{K;\Gamma \vdash_\kappa \lambda x.e\colon \tau_1 \to \tau_2} \qquad \textit{Tk-Appl}\ \frac{K;\Gamma \vdash_\kappa e_1\colon \tau' \to \tau \qquad K;\Gamma \vdash_\kappa e_2\colon \tau'}{K;\Gamma \vdash_\kappa e_1\ e_2\colon \tau}$$

$$\textit{Tk-Pair}\ \frac{K;\Gamma \vdash_\kappa e_1\colon \tau_1 \qquad K;\Gamma \vdash_\kappa e_2\colon \tau_2}{K;\Gamma \vdash_\kappa (e_1, e_2)\colon \tau_1 \times \tau_2}$$

$$\textit{Tk-Tag}\ \frac{K;\Gamma \vdash_\kappa e\colon \tau \qquad K \ni \alpha :: \kappa_\alpha \qquad \kappa_\alpha \vDash (\{\text{`}tag\}, \mathcal{L}, \{\text{`}tag\colon \tau\})}{K;\Gamma \vdash_\kappa \text{`}tag(e)\colon \alpha}$$

$$\textit{Tk-Match}\ \frac{\begin{array}{c} K;\Gamma \vdash_\kappa e_0\colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \\ \forall\, i \in I \qquad K \vdash p_i\colon \tau_0 \Rightarrow \Gamma_i \qquad K;\Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_i) \vdash_\kappa e_i\colon \tau \end{array}}{K;\Gamma \vdash_\kappa \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \tau}$$

FIGURE 3.4 Typing relation.

We use the κ subscript in the turnstile symbol to distinguish this relation from the one for the set-theoretic type system of the next chapter.

The rules are mostly straightforward, with the exception of that for pattern matching, which relies on two other relations that we define and explain below. Here we briefly discuss the other rules.

We embed instantiation and generalization into the rules for variables and pattern matching so that all rules derive types and not type schemes. Hence, *Tk-Var* allows the derivation of any instance of the scheme which Γ binds x to.

The rules for constants, abstractions, applications, and pairs are standard.

Typing variant expressions requires us to consider the kinding environment. Rule *Tk-Tag* states that $\text{`}tag(e)$ can be typed by any variable $\alpha$ such that $\alpha$ has a constrained kind in $K$ which entails the 'minimal' kind for this expression. Specifically – if $\alpha :: (L, U, T) \in K$ – we require $\text{`}tag \in L$ and $\text{`}tag\colon \tau \in T$, where $\tau$ is a type for $e$. Note that $T$ may not assign more than one type to $\text{`}tag$, since $\text{`}tag \in L$.

Typing pattern matching

Typing a pattern-matching expression $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}$ with a type τ should consist of a few different steps:

$$\textit{TPk-Wildcard}\ \frac{}{K \vdash \_ \colon \tau \Rightarrow \varnothing} \qquad \textit{TPk-Var}\ \frac{}{K \vdash x \colon \tau \Rightarrow \{x \colon \tau\}} \qquad \textit{TPk-Const}\ \frac{}{K \vdash c \colon b_c \Rightarrow \varnothing}$$

$$\textit{TPk-Pair}\ \frac{K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma_1 \qquad K \vdash p_2 \colon \tau_2 \Rightarrow \Gamma_2}{K \vdash (p_1, p_2) \colon \tau_1 \times \tau_2 \Rightarrow \Gamma_1 \cup \Gamma_2}$$

$$\textit{TPk-Tag}\ \frac{K \vdash p \colon \tau \Rightarrow \Gamma \qquad K \ni \alpha :: (L, U, T) \qquad \text{`}tag \in U \Rightarrow \text{`}tag \colon \tau \in T}{K \vdash \text{`}tag(p) \colon \alpha \Rightarrow \Gamma}$$

$$\textit{TPk-And}\ \frac{K \vdash p_1 \colon \tau \Rightarrow \Gamma_1 \qquad K \vdash p_2 \colon \tau \Rightarrow \Gamma_2}{K \vdash p_1 \& p_2 \colon \tau \Rightarrow \Gamma_1 \cup \Gamma_2} \qquad \textit{TPk-Or}\ \frac{K \vdash p_1 \colon \tau \Rightarrow \Gamma \qquad K \vdash p_2 \colon \tau \Rightarrow \Gamma}{K \vdash p_1 | p_2 \colon \tau \Rightarrow \Gamma}$$

FIGURE 3.5 Pattern environment generation.

- typing the expression e_0 to be matched with some type τ_0;
- generating a type environment for the capture variables of each pattern;
- typing each branch e_i with type τ, in a type environment updated with that generated by the corresponding pattern and generalized;
- optionally, checking that the patterns are exhaustive, that is, that each possible value of type τ_0 matches at least one pattern;
- optionally, checking non-redundancy, that is, that there are no branches which will never be selected regardless of which value e_0 produces.

The rule *Tk-Match* includes all but the last step as premises. In particular, $K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i$ states that the pattern p_i generates the environment Γ_i when matching the type τ_0 in K. This means that the substitution produced by matching a value of type τ_0 with p_i will replace each capture variable x of p_i with a value of type $\Gamma_i(x)$. The environment is then generalized.

We write the exhaustiveness condition as $\tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$. We do not check non-redundancy, though a partial check is implicit in the definition of $K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i$ since only certain combinations of K, τ_0, and p_i allow us to derive an environment.

The relation for environment generation is defined as follows.

DEFINITION 3.19: Pattern environment generation The environment generated by pattern matching is given by the relation $K \vdash p \colon \tau \Rightarrow \Gamma$ (*the pattern p may match type τ in K, producing the bindings in Γ*), defined by the rules in Figure 3.5. □

We have remarked that some combinations of type and pattern do not allow us to derive any environment. For example, the constant pattern `1` may be used only if τ_0 is `int` and not if it is `bool`. Here, this serves as a weak form of non-redundancy checking (if τ_0 = `bool`, `1` would never be selected); in

OCaml, it is actually necessary for soundness since – due to type erasure – integers and booleans cannot be distinguished at run-time, so comparing them is unsafe. Obviously, we are not eliminating all redundancy: repeated patterns are allowed, and expressions with variant types allow any variant pattern, even those for tags not in their upper bound (this is necessary to prove the stability of typing under type substitutions).

Indeed, it is worth noting how environment generation considers each pattern independently. To construct the environment for a pattern, we might actually take the previous, skipped patterns into consideration to have a more precise result (we will do so in the set-theoretic type system).

We remark also that the environment is uniquely determined by the other arguments in every case except *TPk-Tag*. In that rule, we can make different choices of τ_1 and generate different environments. This means that, if $\backprime tag$ has a conjunctive type (*i.e.* if it has multiple types in T), we can pick any of its types. In a system with intersection types, we might select the intersection itself and not have to project one factor.

We have said that we also require exhaustiveness. OCaml does not – it signals non-exhaustiveness with a warning – but we do so in order to have a simpler statement for soundness and to facilitate the comparison with the system of the next chapter. We give the following definition.

DEFINITION 3.20: Exhaustiveness We say that a set of patterns P is *exhaustive* with respect to a type τ in a kinding environment K, and we write $\tau \preccurlyeq_K P$, when any value that can be typed with any admissible substitutions of τ is accepted by at least one pattern, that is, when

$$\forall K', \theta, v.\ (K \vdash \theta : K' \wedge K'; \varnothing \vdash_{\kappa} v : \tau\theta) \Rightarrow \exists p \in P, \varsigma.\ v/p = \varsigma\,. \qquad \square$$

We consider any admissible instantiation of τ (*i.e.* $\tau\theta$ for any admissible θ) to account for generalization. For example, consider a parametrically-polymorphic abstraction $\lambda x.e$, where x has type α and α is unconstrained in K. Any pattern-matching expression on x which appears in e must be exhaustive with respect to any possible instantiation of α.

We do not discuss how exhaustiveness can be effectively computed. For more information on how OCaml checks it, see Garrigue (2004) and Maranget (2007). Likewise, we do not discuss checking for redundancy. Here these notions cannot be expressed directly at the level of types, which instead becomes becomes possible once we enrich our language of types sufficiently.

Type soundness

We conclude this section by stating the type soundness property of the *VariantsK* type system. It is derived as a corollary of the two properties of progress and subject reduction. Their proofs, as all others, are in Appendix A.

We say that an expression e is well-typed if there exist a K, a Γ, and a τ such that $K; \Gamma \vdash_{\kappa} e : \tau$. If e is closed (*e.g.* if it is a value), we can always take $\Gamma = \varnothing$.

THEOREM 3.1: Progress *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

THEOREM 3.2: Subject reduction *Let e be an expression and τ a type such that $K; \Gamma \vdash_{\kappa} e : \tau$. If $e \rightsquigarrow e'$, then $K; \Gamma \vdash_{\kappa} e' : \tau$.*

COROLLARY 3.3: Type soundness *Let e be a well-typed, closed expression, that is, such that $K; \varnothing \vdash_{\kappa} e : \tau$ holds for some τ. Then, either e diverges or it reduces to a value v such that $K; \varnothing \vdash_{\kappa} v : \tau$.*

3.4 VARIANTS IN OTHER MODELS AND IN OCAML

We discuss here the differences between our system and other calculi and type systems for variants in ML and the reasons for the choices we have made. We also compare our formalization with OCaml itself.

Other formalizations of variants in ML

VariantsK is based on *structural polymorphism*, which is distinct from parametric polymorphism – the usual form of polymorphism in ML – because there are type schemes whose possible instances are restricted by constraints. Type systems which extend the Hindley-Milner type system with structural polymorphism have long been studied with the practical application of describing polymorphic typing for both variants and records – the two being the dual of one another. Such systems have also been described as instances of constraint-based frameworks such as HM(X) (Odersky, Sulzmann, and Wehr, 1999; Pottier and Rémy, 2005).

Our presentation is closely based on the system in Garrigue (2002, 2015). Those works supersede the earlier description in Garrigue (1998) and correspond most closely to the current implementation of OCaml.

The differences in our presentation are mainly three. First, we omit the abstract framework of constraint domains and introduce kinds directly in the form we need for polymorphic variants. Garrigue's system is much more general, since it also covers record typing and, by using different constraint domains, it can describe other systems, such as the simpler form of polymorphism in Ohori (1995) as well as a sort of dependent typing of pattern matching (which is not used in OCaml).

Second, we include full pattern matching, while Garrigue's calculi only include a 'shallow' form of case analysis. Indeed, pattern matching is rarely

formalized in ML-like calculi, though there are exceptions (for instance, see Pottier and Rémy, 2003). The only work on pattern matching on polymorphic variants, to our knowledge, is Garrigue (2004), and it only concerns some aspects of type reconstruction. We have chosen to formalize it for concreteness and to compare the formalization with that based on set-theoretic types; we argue that one of the advantages of the latter system is that it allows a much cleaner formalization and a much more precise typing of matching.

Third, we do not treat type inference. Extending Garrigue's type reconstruction to include full pattern matching in a way that corresponds reasonably well to its implementation in OCaml is quite complex and it is not the goal of our work. Rather, we prefer to study the problem of reconstruction in the context of our set-theoretic type system: since that system will be capable of typing all programs *VariantsK* can type, reconstruction for it should also serve for *VariantsK*. More information on how reconstruction of pattern matching on variants works in OCaml can be found in Garrigue (2004).

Variants, VariantsK, and OCaml

We consider only a very small fragment of OCaml in our formalization and do not deal with type reconstruction at all. However, the proper description of pattern matching has already proven to be somewhat cumbersome, and there are a few aspects in which it does not match the actual implementation in OCaml.

Most notably, we require exhaustiveness. We have already said that it makes soundness easier to state: a non-exhaustive pattern matching can be stuck. In OCaml, an exception is raised in this case; indeed, if we added a notion of dynamic error to our calculus, we might add an unsafe version of match with the same behaviour as that of OCaml. We feel that it is not so necessary in this calculus: though it might be convenient in practice to allow non-exhaustive matching on ordinary variants, for polymorphic ones it is seldom of any use.

There is another difference in the formalization, which concerns a quite technical point in the handling of conjunctive types. We lift a restriction which is not needed for soundness but is imposed in OCaml because it makes reconstruction generate more intuitive types and allows the earlier detection of some errors (as motivated in Garrigue, 2004). Here we avoid it for simplicity. The following function illustrates the difference.

```
fun (x: [< `A of int & bool | `B of int ]) →
  match x with `A n → n >= 1 | `A b → not b | `B _ → true
▸ Error: This expression has type int
    but an expression was expected of type bool
```

OCaml rejects it, while it accepts it if we remove either the first or the second branch: even though the argument of `A has both type int and type bool

(which, incidentally, means it can never reduce to a value) we must use it with the same type in both branches. We allow different branches to choose different types, which seems to be the simplest choice as we do not study reconstruction.

Yet another difference is that we do not model the feature of pattern matching which allows variant types to be refined by listing some tags in an alias pattern, as in the following code.

```
let f: [< `A | `B ] → int = function `A → 1 | `B as y → (function `B → 1) y
▸ [< `A | `B ] → int
```

The precise typing of pattern matching in the set-theoretic system will subsume it. Garrigue (2002) models it as a split construct separate from normal case-analysis.

Finally, we remark that the untyped semantics we have given is not faithful to the implementation of OCaml, though the discrepancies only arise when we consider expression that are ill-typed in OCaml and in *VariantsK*. This is because OCaml performs type erasure and uses the same representation at run-time for values of different types (for instance, false and true are represented as 0 and 1). As a result, matching is only defined when the pattern and the type are 'compatible', while in *Variants* we say matching fails whenever they are not compatible: for example, we have 1/true = Ω, while in OCaml it would be successful were it not blocked by the type system. We address this point in Section 6.3, where we introduce a new semantics in which matching may be undefined.

4 *Variants with set-theoretic types*

We now present *VariantsS*, an alternative type system for the *Variants* calculus introduced in the previous chapter. In this system, we expand the language of types considerably: we add singleton types for constants, types for variants, and the set-theoretic connectives of union, intersection, and negation. Additionally, we employ a subtyping relation, which is *semantic* in that it is defined starting from an interpretation of types as sets. We treat the 'lower-level' details of subtyping only cursorily, since we can readily reuse a system that has been studied elsewhere.

The approach we take is drastically different from that followed in *VariantsK*. There, we add a kinding system to record information that types cannot express: we do so in order to keep the same types as in ML and not to add subtyping, so we can still rely on unification for type reconstruction. Here, conversely, we move all information to the types themselves: the rich language of types and the flexibility of semantic subtyping make this possible. In particular, the representation of variant types employs unions and their subtyping, and it relies on the possibility of encoding bounded quantification in terms of set-theoretic union and intersection.

We argue that *VariantsS* has several advantages with respect to the previous system. It is more expressive: it is able to type some programs that *VariantsK* rejects – while they are actually type-safe – and it can derive more precise types for some programs that *VariantsK* can type too. It is arguably more intuitive as well: apart from the presence of subtyping, typing works much like in ML. Explicit types for variants simplify their typing; handling variant polymorphism with subtyping – instead of instantiation – avoids the loss of polymorphism of variant types bound to λ-abstracted variables rather than let-bound ones.

These advantages are counterbalanced by complications introduced by subtyping, which we mostly ignore in the description of the deductive type system as they are studied elsewhere. The derivation of a typing algorithm and of a type reconstruction system both pose challenges; for the former we refer to previous work, while we study the latter in the next chapter.

The system lends itself particularly well to the typing of pattern matching, which is central to the use of variants in concrete programming. Singleton types and connectives allow us to describe exhaustiveness and irredundancy checking at the level of types, yielding a simpler description. They also allow us to generate more precise environments from patterns.

We have already introduced intuitively the idea of semantic subtyping and the type systems which employ it. *VariantsS* is based on the system in Castagna *et al.* (2014) and Castagna, Nguyễn, Xu, and Abate (2015), which is the foundation of the polymorphic extension of ℂDuce. However, there are significant differences, motivated by the concrete application and by the desire to conform the presentation to standard descriptions of the Hindley-Milner type system. Most significantly, our calculus does not permit type-cases on arrow types, and this allows many simplifications throughout.

This chapter describes the deductive type system. In the last section, we compare it with *VariantsK* – to show it is more expressive – and we discuss the differences we have introduced with respect to the aforementioned type system used in ℂDuce.

4.1 Types and subtyping

In this section, we define the types of the *VariantsS* system and introduce the properties of its subtyping relation.

We assume that there exists a countable set $\mathcal{V}$ of *type variables*, ranged over by $\alpha, \beta, \gamma, \ldots$. We consider the set $\mathcal{C}$ of language constants, ranged over by c, and the set $\mathcal{L}$ of tags, ranged over by $`tag$. We also assume that there exists a finite set $\mathcal{B}$ of *basic types*, ranged over by b.

Definition 4.1: Types A *type* t is a term co-inductively produced by the following grammar:

$t ::= \alpha$		type variable
	$\mid b$	basic
	$\mid c$	constant singleton
	$\mid t \to t$	arrow
	$\mid t \times t$	product
	$\mid `tag(t)$	variant
	$\mid t \vee t$	union
	$\mid \neg t$	negation
	$\mid \mathbb{0}$	empty

which satisfies two additional constraints:

- (*regularity*) the term must have a finite number of different sub-terms;
- (*contractivity*) every infinite branch must contain an infinite number of occurrences of atoms (*i.e.* a type variable or the immediate application of a type constructor: constant, arrow, product, or variant). □

We introduce the following abbreviations.

$$\begin{array}{rcll} t_1 \wedge t_2 & = & \neg(\neg t_1 \vee \neg t_2) & \text{intersection} \\ t_1 \setminus t_2 & = & t_1 \wedge (\neg t_2) & \text{difference} \\ \mathbb{1} & = & \neg\mathbb{0} & \text{any} \end{array}$$

We enrich the language of types from that given in Definition 3.8 in multiple ways. First, we introduce set-theoretic type connectives: union, intersection, and complementation, as well as a top and a bottom type. We also add recursive types by interpreting the grammar co-inductively. Contractivity is imposed to bar out ill-formed types such as $t = t \vee t$ (which does not give any information on the set of values it represents) or $t = \neg t$ (which cannot represent any set of values). These are the essential elements of set-theoretic types as studied in the theory of semantic subtyping in Frisch, Castagna, and Benzaken (2008); Castagna and Xu (2011) then add type variables to provide parametric polymorphism.

Second, we add singleton types for constants: for example, we have a type true for the constant true, which is more precise than the corresponding basic type bool. These are necessary for the precise typing of pattern matching.

Finally, we add explicit types for variants, rather than employing type variables (which we use only for parametric polymorphism). These types have the form $\texttt{`}tag(t)$: the type of variant expressions with tag $\texttt{`}tag$ and an argument of type t. Type connectives allow us to represent all variant types of *VariantsK* by combining types of this form, as we describe in detail below.

VARIANT TYPES AND BOUNDED QUANTIFICATION *VariantsK* uses variables to type variants, but these variables have kinds attached to them and, when we quantify them in a type scheme, these kinds constrain the possible instantiations of the scheme. This is conceptually a form of bounded quantification: a variable of kind (L, U, T) may only be instantiated by other variables which fall into the bounds – the lower bound being determined by L and T, the upper one by U and T.

These bounds can be represented in our system as unions of variant types $\texttt{`}tag(t)$. For instance, consider in *VariantsK* a constrained variable $\alpha$ of kind $(\{\texttt{`A}\}, \{\texttt{`A}, \texttt{`B}\}, \{\texttt{`A}: \mathsf{bool}, \texttt{`B}: \mathsf{int}\})$. If we quantify α, we can then instantiate it with variables whose kinds entail that of α. Using our variant types and unions, we write the lower bound as $t_\mathrm{L} = \texttt{`A}(\mathsf{bool})$ and the upper one as $t_\mathrm{U} = \texttt{`A}(\mathsf{bool}) \vee \texttt{`B}(\mathsf{int})$. In our system, α should be a variable with bounded quantification, which can only be instantiated by types t such that $t_\mathrm{L} \leq t \leq t_\mathrm{U}$.

However, we do not need to introduce bounded quantification as a feature of our language: we can use type connectives to encode it. The possible instantiations of α (with the bounds above) and the possible instantiations of $(t_\mathrm{L} \vee \beta) \wedge t_\mathrm{U}$, with no bound on β, are equivalent. We use the latter form: we internalize the bounds in the type itself by union and intersection. In this way, we need no system of constraints extraneous to types.

TOP TYPES Note that we can define top types for each 'family' of types: constants, arrows, products, and variants. They are disjoint and each value belongs to exactly one of these. The top types are defined as follows.

$$\begin{aligned}
\mathbb{1}_{\mathrm{B}} &= \bigvee_{b \in \mathcal{B}} b && \text{constants} \\
\mathbb{1}_{\mathrm{A}} &= \mathbb{0} \to \mathbb{1} && \text{arrows} \\
\mathbb{1}_{\mathrm{P}} &= \mathbb{1} \times \mathbb{1} && \text{pairs} \\
\mathbb{1}_{\mathrm{V}} &= \neg(\mathbb{1}_{\mathrm{B}} \vee \mathbb{1}_{\mathrm{A}} \vee \mathbb{1}_{\mathrm{P}}) && \text{variants}
\end{aligned}$$

The type $t_1 \to t_2$ is that of functions which, if they are given an argument in t_1 and they do not diverge, yield a result in t_2. Hence, $\mathbb{0} \to t$ is actually equivalent to $\mathbb{0} \to \mathbb{1}$ for any t, as any of them is the type of all functions: no value is in $\mathbb{0}$, hence we never have any constraint on the result.

Conversely, $\mathbb{1} \to \mathbb{0}$ is the type of functions that (provably) diverge on all inputs: a function of this type should yield a value in the empty type whenever it terminates, and that is impossible.

ENCODING OF VARIANT TYPES Variant types are the only addition we make to the types used in previous work on semantic subtyping with parametric polymorphism (Castagna and Xu, 2011) and on a calculus and type system for polymorphic functions with set-theoretic types (Castagna *et al.*, 2014; Castagna, Nguyễn, Xu, and Abate, 2015).

Indeed, we could choose to treat variants as syntactic sugar, by adding singleton types for tags and encoding a variant type $`tag(t)$ as the pair type $`tag \times t$. In the design of a language we might actually represent variants as pairs in this way, in the semantics and not just for typing. This would allow us, for instance, to match any variant value with the pattern (t, a) to capture its tag in t and its argument in a.[1]

We could adopt this encoding explicitly here and have $`tag(t)$ be just an abbreviation. However, our syntax and semantics (and indeed those of OCaml) treat pairs and variants as distinct sorts of expressions: notably, a variant value does *not* match (x, y). For consistency with this semantics, we have chosen to treat their types as distinct families too. Otherwise, we would have to distinguish between 'ordinary' product types and those which encode variants in some cases, complicating a few definitions.

Thus we introduce a new constructor for variant types. However, we still use this encoding at a lower level: we derive from it the properties of subtyping for variant types. An analogous solution is adopted in ℂDuce for XML types: there is a distinct constructor whose subtyping is defined by an encoding into product types.

1 In fact, this encoding is used in the interface between ℂDuce and OCaml and in general to emulate variants in ℂDuce.

Subtyping

There exists a *subtyping* relation between types. We write $t_1 \leq t_2$ when t_1 is a subtype of t_2. We write $t_1 \simeq t_2$ when t_1 and t_2 are equivalent with respect to subtyping, that is, when $t_1 \leq t_2$ and $t_2 \leq t_1$. The definition and properties of this relation are studied in Castagna and Xu (2011) – except for variant types, which can be added by encoding them – and we do not report them here. We only give a brief account of such properties as we will use later on.

Subtyping has a semantic definition, in the sense that $t_1 \leq t_2$ holds if and only if $[\![t_1]\!] \subseteq [\![t_2]\!]$, where $[\![\cdot]\!]$ is an interpretation function mapping types to sets of elements from some domain $\mathcal{D}$. We can think of $\mathcal{D}$ as the set of all values in the language, so that types are interpreted as sets of values.

The properties of connectives with respect to subtyping are given by a few conditions on the interpretation, which make it 'set-theoretic'. Union types must be interpreted into unions and negation into complementation with respect to $\mathcal{D}$. The bottom type is interpreted as the empty set. Basic types are interpreted as sets of constants, products as Cartesian products. An arrow type $t_1 \to t_2$ is interpreted equivalently – as far as subtyping is concerned – to the set of relations f where, for each pair $(x, y) \in f$, if x is in the interpretation of t_1 then y is in the interpretation of t_2.

The main contribution of Castagna and Xu (2011) is the definition of semantic subtyping for types containing variables. Intuitively, a variable is only a subtype of itself and of $\mathbb{1}$. Thus, if it occurs in covariant position in a type, it can only be subsumed to $\mathbb{1}$ or to unions where it appears explicitly. Conversely, if it occurs in contravariant position, it can be subsumed to $\mathbb{0}$ or to intersections where it appears explicitly. Subtyping is preserved by type substitutions, that is, $t_1 \leq t_2$ implies $t_1\theta \leq t_2\theta$ for any type substitution θ.

A consequence of this semantic definition is that pairs and variants with the empty type as sub-terms are themselves empty. For example, $\mathsf{int} \times \mathbb{0}$ is equivalent to $\mathbb{0}$, because the Cartesian product $A \times \varnothing$ is empty for any A: we cannot build a pair value whose second component is in the empty type.

4.2 TYPE SYSTEM

We move to the description of typing in *VariantsS*. As mentioned earlier, we strive to keep the presentation as similar to *VariantsK* as possible. Kinds and kinding environments are no longer needed because all their information can be represented by the types themselves: this shortens many definitions.

We begin by introducing type schemes, defined as in ML.

DEFINITION 4.2: Type schemes A *type scheme* s is of the form $\forall A.\, t$, where A is a finite set $\{\alpha_1, \ldots, \alpha_n\}$ of type variables.

We identify a type scheme $\forall \varnothing.\, t$, which quantifies no variable, with the type t itself: thus, we view types as a subset of type schemes. Furthermore,

$$
\begin{aligned}
\alpha\theta &= \begin{cases} t' & \text{if } {}^{t'}/_{\alpha} \in \theta \\ \alpha & \text{otherwise} \end{cases} \\
b\theta &= b \\
c\theta &= c \\
(t_1 \to t_2)\theta &= (t_1\theta) \to (t_2\theta) \\
(t_1 \times t_2)\theta &= (t_1\theta) \times (t_2\theta) \\
\text{`}tag(t)\theta &= \text{`}tag(t\theta) \\
(t_1 \vee t_2)\theta &= (t_1\theta) \vee (t_2\theta) \\
(\neg t)\theta &= \neg(t\theta) \\
\mathbb{0}\theta &= \mathbb{0}
\end{aligned}
$$

FIGURE 4.1 Application of a type substitution to a type.

we consider type schemes up to renaming of the variables they bind, and we disregard useless quantification (*i.e.* quantifying variables which do not occur in the type). □

We introduce the set of free variables occurring in a type or type scheme, without giving the complete definition. As types are co-inductive, it can given by memoization (see Castagna *et al.*, 2014, Definition A.2).

DEFINITION 4.3: Free variables We define $var(t)$ to be the set of type variables occurring in a type t. We say they are the *free variables* of t, and we say that t is *ground* or *closed* if and only if $var(t)$ is empty.

We extend the definition to type schemes as $var(\forall A.\, t) = var(t) \setminus A$. □

We define type substitutions and their application to types. To account for types being recursive, we give the definition by co-induction.

DEFINITION 4.4: Type substitutions A *type substitution* θ is a finite mapping of type variables to types. We write $[\, {}^{t_i}/_{\alpha_i} \mid i \in I \,]$ for the type substitution which simultaneously replaces α_i with t_i, for each $i \in I$. We write $t\theta$ for the application of the substitution to a type t; application is co-inductively defined by the equations in Figure 4.1.

We extend the $var(\cdot)$ operation to substitutions as

$$var(\theta) = \bigcup_{\alpha \in dom(\theta)} var(\alpha\theta)\,.$$

We extend application of substitutions to type schemes $\forall A.\, t$. By renaming quantified variables, we assume $A \cap (dom(\theta) \cup var(\theta)) = \varnothing$, and we have $(\forall A.\, t)\theta = \forall A.\, t\theta$. □

We define type environments and the operations of generalization of types and instantiation of type schemes. The order of generality between type schemes must now account for subtyping.

DEFINITION 4.5: Type environments A *type environment* Γ is a partial mapping from expression variables to type schemes. We write type environments as $\Gamma = \{x_1 : s_1, \ldots, x_n : s_n\}$.

We write Γ, Γ' for the updating of the type environment Γ with the new bindings in Γ'. It is defined as follows.

$$(\Gamma, \Gamma')(x) = \begin{cases} \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma') \\ \Gamma(x) & \text{otherwise} \end{cases}$$

We extend the $\mathit{var}(\cdot)$ operation to type environments as

$$\mathit{var}(\Gamma) = \bigcup_{s \in \mathit{range}(\Gamma)} \mathit{var}(s) \,. \qquad \square$$

DEFINITION 4.6: Generalization We define the *generalization* $\mathit{gen}_\Gamma(t)$ of a type t with respect to a type environment Γ as the type scheme

$$\mathit{gen}_\Gamma(t) = \forall A.\, t$$

where $A = \mathit{var}(t) \setminus \mathit{var}(\Gamma)$.

We extend this definition to type environments which only contain types (*i.e.* trivial type schemes) as

$$\mathit{gen}_\Gamma(\{x_1 : t_1, \ldots, x_n : t_n\}) = \{x_1 : \mathit{gen}_\Gamma(t_1), \ldots, x_n : \mathit{gen}_\Gamma(t_n)\} \,. \qquad \square$$

DEFINITION 4.7: Instances of a type scheme We define the *instances* of a type scheme $\forall A.\, t$ as the following set of types:

$$\mathit{inst}(\forall A.\, t) = \{\, t\theta \mid \mathit{dom}(\theta) \subseteq A \,\} \,.$$

We say that a type scheme s_1 is *more general* than a type scheme s_2, and we write $s_1 \sqsubseteq s_2$, if for every $t_2 \in \mathit{inst}(s_2)$ there exists a $t_1 \in \mathit{inst}(s_1)$ such that $t_1 \leq t_2$.

We extend this notion to type environments as

$$\Gamma_1 \sqsubseteq \Gamma_2 \iff \mathit{dom}(\Gamma_1) = \mathit{dom}(\Gamma_2) \wedge \forall x \in \mathit{dom}(\Gamma_1).\, \Gamma_1(x) \sqsubseteq \Gamma_2(x) \,. \qquad \square$$

We now define the typing relation itself. As before, we present the definitions related to pattern matching afterwards.

DEFINITION 4.8: Typing relation The typing relation $\Gamma \vdash_s e : t$ (*e is given type t in the type environment* Γ) is defined by the rules in Figure 4.2. $\square$

$$\textit{Ts-Var}\ \frac{t \in \mathit{inst}(\Gamma(x))}{\Gamma \vdash_{\mathrm{s}} x\colon t} \qquad \textit{Ts-Const}\ \frac{}{\Gamma \vdash_{\mathrm{s}} c\colon c}$$

$$\textit{Ts-Abstr}\ \frac{\Gamma, \{x\colon t_1\} \vdash_{\mathrm{s}} e\colon t_2}{\Gamma \vdash_{\mathrm{s}} \lambda x.e\colon t_1 \to t_2} \qquad \textit{Ts-Appl}\ \frac{\Gamma \vdash_{\mathrm{s}} e_1\colon t' \to t \qquad \Gamma \vdash_{\mathrm{s}} e_2\colon t'}{\Gamma \vdash_{\mathrm{s}} e_1\ e_2\colon t}$$

$$\textit{Ts-Pair}\ \frac{\Gamma \vdash_{\mathrm{s}} e_1\colon t_1 \qquad \Gamma \vdash_{\mathrm{s}} e_2\colon t_2}{\Gamma \vdash_{\mathrm{s}} (e_1, e_2)\colon t_1 \times t_2} \qquad \textit{Ts-Tag}\ \frac{\Gamma \vdash_{\mathrm{s}} e\colon t}{\Gamma \vdash_{\mathrm{s}} \text{`}\mathit{tag}(e)\colon \text{`}\mathit{tag}(t)}$$

$$\textit{Ts-Match}\ \frac{\Gamma \vdash_{\mathrm{s}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lfloor p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor \qquad \forall i \in I\ \begin{cases} t'_i = \mathbb{0} & \text{if } t_i \leq \mathbb{0} \\ \Gamma, \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_{\mathrm{s}} e_i\colon t'_i & \text{otherwise} \end{cases}}{\Gamma \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \bigvee_{i\in I} t'_i}$$

$$\textit{Ts-Subsum}\ \frac{\Gamma \vdash_{\mathrm{s}} e\colon t' \qquad t' \leq t}{\Gamma \vdash_{\mathrm{s}} e\colon t}$$

FIGURE 4.2 Typing relation.

We use the s subscript in the turnstile symbol to distinguish this typing relation from that of *VariantsK*.

We set aside the rule for pattern-matching expressions for the moment and present the others. They are all straightforward and very close to those of *VariantsK*. Variables, abstractions, applications, and pairs are typed as in ML. Constants are typed with their singleton type. The rule for variant expression is simplified since we have explicit types for them.

The significant difference, of course, is the presence of the subsumption rule *Ts-Subsum*, by which we can assign to an expression a supertype of any type it has. It is the only rule that is not syntax-directed.

Typing pattern matching

Let us examine the rule *Ts-Match*. As in *VariantsK*, we require the expression e_0 to be matched to have some type t_0. The subtyping condition $t_0 \leq \bigvee_{i\in I} \lfloor p_i \rfloor$ formalizes exhaustiveness, as we will see shortly. Unlike the previous system, we do not require every branch to be well-typed. Rather, we disregard those branches which we know will never be selected: we compute an input type t_i for each branch which is empty if the branch is useless.

We then define an output type t'_i for each branch, which is $\mathbb{0}$ for useless branches (as they will never produce a result) and the type assigned to e_i for

$$
\begin{aligned}
\lbag \_ \rbag &= \mathbb{1} \\
\lbag x \rbag &= \mathbb{1} \\
\lbag c \rbag &= c \\
\lbag (p_1, p_2) \rbag &= \lbag p_1 \rbag \times \lbag p_2 \rbag \\
\lbag `tag(p) \rbag &= `tag(\lbag p \rbag) \\
\lbag p_1 \& p_2 \rbag &= \lbag p_1 \rbag \wedge \lbag p_2 \rbag \\
\lbag p_1 | p_2 \rbag &= \lbag p_1 \rbag \vee \lbag p_2 \rbag
\end{aligned}
$$

FIGURE 4.3 Accepted type of a pattern.

the others. To type each branch, we generate an environment $t_i /\!/ p_i$, which is then generalized. The final type for the expression is the union of all t'_i: union types are useful here, as different branches can have different types.

Set-theoretic types are beneficial most of all because of the following property: the set of values accepted by a pattern is always described precisely by a type. Clearly, the existence of constant patterns means we need singleton types for this to be true: no type of *VariantsK* corresponds exactly to the set of values matched by true, for instance. Since we have union and intersection patterns, we also need the corresponding connectives. Wildcards and variables require a top type. Having all these, we can give the following definition.

DEFINITION 4.9: Accepted type of a pattern The *accepted type* $\lbag p \rbag$ of a pattern p is defined inductively by the equations in Figure 4.3. □

This definition is given by induction on the type, but it is equivalent to the semantic definition which defines the accepted type of a pattern p as the set of values accepted by p. Thus we know that a value matches a pattern if and only if it can be typed with the accepted type of that pattern.

This allows us to represent exhaustiveness with our types. The union $\bigvee_{i \in I} \lbag p_i \rbag$ is the set of values which will be accepted by one pattern at least. We require $t_0 \leq \bigvee_{i \in I} \lbag p_i \rbag$, that is, that any value which can be produced by e_0 be accepted by a pattern.

Non-redundancy of a pattern can also be expressed in terms of accepted types. The type t_i we define for each branch corresponds exactly to the values that will be matched by the pattern p_i: $t_0 \setminus \bigvee_{j<i} \lbag p_j \rbag$ is the subtype of t_0 corresponding to the values that are not accepted by the previous patterns; by intersecting it with $\lbag p_i \rbag$, we only consider those that p_i will accept.

If $t_i \leq \mathbb{0}$, we know the branch will never be selected: any value t_0 can produce will either be matched by one of the preceding patterns or fall through to the following ones. Here, we choose to ignore the corresponding

branch: we do not consider it in the final type, and we do not type it at all. Even if it were ill-typed, it could never be reached: hence it is sound to accept it. Alternatively, we might require $t_i \not\leq \mathbb{0}$ to hold and thus forbid redundant patterns altogether.

Environment generation uses t_i as well, making the resulting environment more precise. For instance, the environment we generate for the pattern x is not $\{x\colon t_0\}$ but rather $\{x\colon t_i\}$ where t_i is a subtype of t_0. We bind the variable not to the type of all values the matched expression may produce, but to the type of the values which can actually reach and match that pattern.

For environment generation, type connectives also introduce a complication. It is no longer true that every product type (a type t such that $t \leq \mathbb{1} \times \mathbb{1}$) is of the form $t_1 \times t_2$: for example, it might be $(\mathsf{int} \times \mathsf{int}) \vee (\mathsf{bool} \times \mathsf{bool})$. Being able to distinguish this type from $(\mathsf{int} \vee \mathsf{bool}) \times (\mathsf{int} \vee \mathsf{bool})$ is useful; however, it means it is no longer trivial to compute the first or second component of a product type.

We introduce two operators for this purpose below, though we do not give their definition – we only state those of their properties which we need in our proofs. See Castagna *et al.* (2014, Appendix C.2.1) for the full details. Variant types are analogous to products in this, so we introduce an operator for them as well (it is just π_2 with the aforementioned encoding of variants as pairs).

PROPERTY 4.10: Projections of product types There exist two functions $\pi_1(\cdot)$ and $\pi_2(\cdot)$ which, given a type $t \leq \mathbb{1} \times \mathbb{1}$, yield types $\pi_1(t)$ and $\pi_2(t)$ such that:

- $t \leq \pi_1(t) \times \pi_2(t)$;
- if $t \leq t_1 \times t_2$, then $\pi_i(t) \leq t_i$;
- if $t \leq t' \leq \mathbb{1} \times \mathbb{1}$, then $\pi_i(t) \leq \pi_i(t')$;
- for all type substitutions θ, $\pi_i(t\theta) \leq \pi_i(t)\theta$. □

PROPERTY 4.11: Projections of variant arguments For every tag $`tag$ there exists a function $\pi_{`tag}(\cdot)$ which, given a type $t \leq `tag(\mathbb{1})$, yields a type $\pi_{`tag}(t)$ such that:

- $t \leq `tag(\pi_{`tag}(t))$;
- if $t \leq `tag(t')$, then $\pi_{`tag}(t) \leq t'$;
- if $t \leq t' \leq `tag(\mathbb{1})$, then $\pi_{`tag}(t) \leq \pi_{`tag}(t')$;
- for all type substitutions θ, $\pi_{`tag}(t\theta) \leq \pi_{`tag}(t)\theta$. □

We now define the generation of an environment from a type and a pattern, using projections to extract product components and variant arguments.

DEFINITION 4.12: Pattern environment generation Given a pattern p and a type $t \leq \lbag p \rbag$, the type environment $t /\!/ p$ generated by pattern matching is defined inductively by the equations in Figure 4.4. □

$$
\begin{array}{lcl}
t//\_ & = & \varnothing \\
t//x & = & \{x\colon t\} \\
t//c & = & \varnothing \\
t//(p_1, p_2) & = & \pi_1(t)//p_1 \cup \pi_2(t)//p_2 \\
t//\backprime tag(p) & = & \pi_{\backprime tag}(t)//p \\
t//p_1 \& p_2 & = & t//p_1 \cup t//p_2 \\
t//p_1 | p_2 & = & (t \wedge \lbag p_1 \rbag)//p_1 \curlyvee (t \setminus \lbag p_1 \rbag)//p_2 \\
 & & \text{where } (\Gamma \curlyvee \Gamma')(x) = \Gamma(x) \vee \Gamma'(x)
\end{array}
$$

FIGURE 4.4 Pattern environment generation.

Intersection patterns allow us to define this as a function, while it had to be a relation in *VariantsK* because of conjunctive types in variant arguments (here, conjunctive types are just intersections in the argument of the variant).

Type soundness

We state here the soundness property of the *VariantsS* type system. As usual, we express it as a corollary of the two properties of progress and subject reduction.

In the following, we say that an expression e is well-typed if there exists a Γ and a t such that $\Gamma \vdash_s e\colon t$. If e is closed (*e.g.* if it is a value), then we can always take $\Gamma = \varnothing$.

THEOREM 4.1: Progress *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

THEOREM 4.2: Subject reduction *Let e be an expression and t a type such that $\Gamma \vdash_s e\colon t$. If $e \rightsquigarrow e'$, then $\Gamma \vdash_s e'\colon t$.*

COROLLARY 4.3: Type soundness *Let e be a well-typed, closed expression, that is, such that $\varnothing \vdash_s e\colon t$ holds for some t. Then, either e diverges or it reduces to a value v such that $\varnothing \vdash_s v\colon t$.*

In the proofs we use the following definition of *freshness* of type variables.

DEFINITION 4.13: Freshness We say that a type variable α is *fresh* with respect to a set of type variables A, and write $\alpha \sharp A$, if $\alpha \notin A$. We extend this definition to types, type environments, and substitutions, by defining $\alpha \sharp t$ as $\alpha \sharp var(t)$, $\alpha \sharp \Gamma$ as $\alpha \sharp var(\Gamma)$, and $\alpha \sharp \theta$ as $\alpha \sharp (dom(\theta) \cup var(\theta))$.

We extend this definition to speak of the freshness of a set of variables, where $A \sharp A'$ means $\forall \alpha \in A.\ \alpha \sharp A'$. □

Algorithmic typing

We have described a deductive type system for *VariantsS*, which includes a subsumption rule that is not syntax-directed. A corresponding typing algorithm can be derived as shown in Castagna *et al.* (2014) and Castagna, Nguyễn, Xu, and Abate (2015), assuming that all abstractions are annotated with their arrow type.

The first of those articles discusses the typing algorithm for a calculus with explicit type substitutions – one wherein, to apply a polymorphic function to an argument, we must write the instantiation explicitly, as in

$$(\lambda^{\alpha\to\alpha} x.x)[^{\mathsf{int}}/_{\alpha}]\ 2\,.$$

The second article covers the inference of these instantiations. Inference of type substitution is not proven to be complete in the presence of overloaded functions (typed with intersections of arrow types), but it is here since we do not allow them. It relies on the algorithm to solve the *tallying problem*, which we discuss in the next chapter.

The appendices of those articles extend the system to add product types. They present an algorithm which is complete and another which renounces completeness for the sake of efficiency (it does not infer type substitutions inside pair projections) but should still be powerful enough in practice.

4.3 Comparison with other systems

We discuss here the relationship between the two type systems we have presented. We wish to show that *VariantsS* is *complete* with respect to *VariantsK*: any program that is well-typed in the latter is also well-typed in the former. This would imply that, by adopting the new type system for our calculus, we extend our language conservatively in the sense that we do not refuse any of the programs we accepted before. Proving this is our first goal here, though we present the proof only for a somewhat restricted case.

The opposite property – that any well-typed program in *VariantsS* be well-typed in *VariantsK* as well – clearly should *not* hold, since the new type system is more expressive. We give below some examples of this greater expressiveness.

Finally, we also compare *VariantsS* with the type system used in ℂDuce, which we have taken as our starting point, to discuss the reasons for the changes we have made.

From now on, when comparing *VariantsK* and *VariantsS*, we use the prefixes *k-* and *s-* to distinguish types, environments, and other notions which exist in both systems. In the definitions, we have also used different

metavariables for types and type schemes: τ and σ for k-types and k-type schemes, t and s for s-types and s-type schemes.

Completeness with respect to VariantsK

To show that *VariantsS* is complete with respect to *VariantsK*, we define a translation of k-types to s-types; this translation naturally depends on a kinding environment to make sense of type variables. We then prove that, whenever an expression may be assigned some type in *VariantsK* (in a certain environment), it may be given the translation of that type in *VariantsS* (in the translated environment).

We work under an assumption which restricts *VariantsK* to simplify the proof: we require kinding environments to be non-recursive. Under this restriction, our system can no longer express recursive variant types (nor any other recursive type); for example, it cannot express the type of integer lists as α where α has kind $(\{\text{`Nil}, \text{`Cons}\}, \{\text{`Nil}, \text{`Cons}\}, \{\text{`Nil: unit}, \text{`Cons: int} \times \alpha\})$. We impose this restriction in order to use induction – with a particular measure of a type in a kinding environment – in the proof of completeness.

Note that recursive types *can* be expressed in *VariantsS*, since the grammar of types is interpreted co-inductively. If we write these types using a μ binder for recursion variables, the type above is $\mu X.\text{`Nil}(\text{unit}) \vee \text{`Cons}(\text{int} \times X)$. We conjecture that completeness holds also in the presence of recursive kinds, but the proof would require us to use co-inductive techniques and enter into much more detail about the definition of subtyping.

We first define this restriction and our measure. Then we define the translation of types, by induction on the measure.

DEFINITION 4.14: Non-recursive kinding environments We say that a kinding environment K is *non-recursive* if, for all $\alpha :: (L, U, T) \in K$, we have $\alpha \notin \bigcup_{tag:\tau \in T} var_K(\tau)$. □

DEFINITION 4.15 We define a function w which, given a k-type τ in a non-recursive kinding environment K, yields the *measure* $w(\tau, K)$ of τ in K. It is defined by the following equations.

$$\begin{aligned} w(\alpha, K) &= \begin{cases} 1 + \sum_{tag:\tau \in T} w(\tau, K) & \text{if } \alpha :: (L, U, T) \in K \\ 1 & \text{otherwise} \end{cases} \\ w(b, K) &= 1 \\ w(\tau_1 \to \tau_2, K) &= w(\tau_1, K) + w(\tau_2, K) + 1 \\ w(\tau_1 \times \tau_2, K) &= w(\tau_1, K) + w(\tau_2, K) + 1 \end{aligned}$$ □

DEFINITION 4.16: Translation of types Let $[\![\cdot]\!]_{(\cdot)}$ be a *type translation function* which, given a k-type τ in a non-recursive kinding environment K, yields an s-type $[\![\tau]\!]_K$. It is defined inductively by the rules in Figure 4.5.

$$
\begin{aligned}
[\![\alpha]\!]_K &= \begin{cases} \alpha & \text{if } \alpha :: \bullet \in K \\ (\mathit{low}_K(L,T) \vee \alpha) \wedge \mathit{upp}_K(U,T) & \text{if } \alpha :: (L,U,T) \in K \end{cases} \\
&\quad \text{where } \mathit{low}_K(L,T) = \bigvee_{\text{`}tag \in L} \text{`}tag(\textstyle\bigwedge_{\text{`}tag:\tau \in T} [\![\tau]\!]_K) \\
&\quad \mathit{upp}_K(U,T) = \begin{cases} \bigvee_{\text{`}tag \in U} \text{`}tag(\textstyle\bigwedge_{\text{`}tag:\tau \in T} [\![\tau]\!]_K) & \text{if } U \neq \mathcal{L} \\ \bigvee_{\text{`}tag \in \mathit{dom}(T)} \text{`}tag(\textstyle\bigwedge_{\text{`}tag:\tau \in T} [\![\tau]\!]_K) & \text{if } U = \mathcal{L} \\ \quad \vee \quad (\mathbb{1}_{\mathrm{v}} \setminus \bigvee_{\text{`}tag \in \mathit{dom}(T)} \text{`}tag(\mathbb{1})) & \end{cases} \\
[\![b]\!]_K &= b \\
[\![\tau_1 \to \tau_2]\!]_K &= [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K \\
[\![\tau_1 \times \tau_2]\!]_K &= [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K
\end{aligned}
$$

FIGURE 4.5 Translation of k-types to s-types.

We extend the translation to type schemes as $[\![\forall A.\, K' \triangleright \tau]\!]_K = \forall A.\, [\![\tau]\!]_{K,K'}$ and to type environments by translating each type scheme pointwise. □

The only complex case in the definition of the translation is that of constrained variables, where we encode the bounded quantification as we have described at the beginning of this chapter. When $\alpha :: (L, U, T) \in K$, we produce the type that corresponds to a variable bounded by a lower and an upper bound. Both bounds are essentially unions of variant types (where empty unions are defined to be $\mathbb{0}$). The lower one is just a union of this form: the argument of each variant is the translation of the type associated to its tag in T. The inner intersection handles conjunctive types, but these do not actually occur for tags in L. The upper bound is analogous when $U \neq \mathcal{L}$; here intersections may have multiple factors.

Finally, when $U = \mathcal{L}$, the upper bound should include any possible tag; we cannot express it as a union of all tags because only finite unions are expressible with our types. Thus, we split into a union of two summands: a part corresponding to the tags mentioned in T – a finite amount – and another part for the others. The former is built normally. The latter is intuitively the infinite union of the types $\text{`}tag(\mathbb{1})$ for each $\text{`}tag \notin \mathit{dom}(T)$: it is the top type of variants, which is 'equivalent' to the infinite union of all types $\text{`}tag(\mathbb{1})$, minus the types for the tags which appear in T.

The equations defining this translation could be interpreted co-inductively to account for recursion; then, they would generate recursive types like $\mu X.\, \text{`}\mathsf{Nil}(\mathsf{unit}) \vee \text{`}\mathsf{Cons}(\mathsf{int} \times X)$ in the example above.

Given this translation, the completeness of *VariantsS* with respect to *VariantsK* is expressed by the following property.

THEOREM 4.4: Preservation of typing *Let e be an expression, K a non-recursive kinding environment, Γ a k-type environment, and τ a k-type. If $K; \Gamma \vdash_{\kappa} e : \tau$, then $[\![\Gamma]\!]_K \vdash_{\mathrm{s}} e : [\![\tau]\!]_K$.*

In addition to assuming kinding environments to be non-recursive, we also make a modification to the system we use in the proofs: we add recursive functions to the language. We do so by adding a fixed-point combinator to the expressions, together with a reduction rule and typing rules for both systems (we give standard definitions in appendix). This is not very restrictive since, in practice, we would want to extend these systems with recursive functions anyway. In this extended system, all arrow types are inhabited by the never-terminating function, which simplifies the proof.

SIMPLIFYING VARIANT TYPES While the translation of constrained variables given above is the general one, the resulting types can be simplified in many cases – just like OCaml does by omitting the variable and printing its kind alone. The simplification, in our case, consists in replacing $(\mathit{low}_K(L, T) \vee \alpha) \wedge \mathit{upp}_K(U, T)$ with $\mathit{low}_K(L, T)$ if it appears in covariant position in a type (*e.g.* in the codomain of an arrow type) or with $\mathit{upp}_K(U, T)$ if it appears in contravariant position (*e.g.* in its domain). This can be done only if the variable always appears with the same variance in the types of the program (in particular, it can be done if it appears only once) and not otherwise.

The justification for this simplification is the following. Assume the variable appears only once. Then, we can choose its instantiation freely, affecting only the type it appears in and no other. If its only occurrence is in covariant position, its instantiation with $\mathbb{0}$ can be seen as the 'canonical' choice because any other instantiation can be obtained by subsumption: we can pick that instantiation because it is more general (in the sense of subtyping) than all others. The resulting type $(\mathit{low}_K(L, T) \vee \mathbb{0}) \wedge \mathit{upp}_K(U, T)$ is equivalent to $\mathit{low}_K(L, T)$ because $\mathit{low}_K(L, T) \leq \mathit{upp}_K(U, T)$. The opposite reasoning leads to choosing $\mathbb{1}$ to instantiate variables that appear in contravariant position. We follow the same idea if a variable appears multiple times, as long as it always has the same variance: the instantiations with $\mathbb{0}$ or $\mathbb{1}$ can reach any other by subsumption.

Increased expressiveness

With respect to *VariantsK*, *VariantsS* is more expressive mainly in three respects: *i*) the use of subsumption rather than instantiation for variant polymorphism avoids the loss of polymorphism for λ-abstracted variables; *ii*) we type pattern matching more precisely, generating more precise environments from patterns and excluding redundant branches; *iii*) we can express more

types and thus describe function behaviour more precisely without losing flexibility (thanks to subtyping). In particular, this system allows us to type all problematic examples given at the end of Section 2.2.

Using subtyping, we can type expression like

```
(fun x → ([ x; `B true ], [ x; `B 2 ])) (`A 1)
```

because we give the type $\texttt{`A}(1)$ to x and then use subsumption to type the two lists. This also avoids the issues with the function

```
let id x = match x with `A|`B → x
▸ ([< `A | `B ] as α) → α
```

which accumulates too many constraints on its result, making it lose polymorphism. In *VariantsS*, id can be given type

$$\alpha \wedge (\texttt{`A}(\texttt{unit}) \vee \texttt{`B}(\texttt{unit})) \to \alpha \wedge (\texttt{`A}(\texttt{unit}) \vee \texttt{`B}(\texttt{unit}))$$

that is, $\forall \alpha \leq (\texttt{`A}(\texttt{unit}) \vee \texttt{`B}(\texttt{unit})).\ \alpha \to \alpha$ with the syntax of bounded quantification. It gives us the same static information as the type in *VariantsK* (*i.e.* we can still give type $\texttt{`A}(\texttt{unit})$ to id `A), but we can combine id `A with any other variant type by subsumption.

Our typing of pattern matching is more precise than that of *VariantsK* and allows us to type the examples we have described on page 13. The improvement relies on being able to represent exactly the set of values which will be matched by a pattern as a type: we use this both to generate more precise environments and to express non-redundancy as a subtyping check (which determines whether we consider a branch or not).

Finally, we can give more precise types to functions defined by pattern matching, which avoids the difficulties in type reconstruction mentioned on page 15 and similar ones. For example, consider the following function.

```
let f = function (true, `A) → 1 | (true, `B) → 2 | (false, _) → 3
▸ bool * [< `A | `B ] → int
```

In OCaml, to avoid non-exhaustiveness we must select the type above. However, the last pattern can actually accept other variants as well. Here, we can give a more precise type for the domain, such as

$$\big(\texttt{true} \times \texttt{`A}(\texttt{unit})\big) \vee \big(\texttt{true} \times \texttt{`B}(\texttt{unit})\big) \vee \big(\texttt{false} \times \mathbb{1}\big)\,.$$

This is also useful when we deal with non-variant types; in general it allows us to give types that ensure matching will be exhaustive without having to make too many patterns redundant or unduly restricting the values they can accept.

Introducing intersections and using semantic subtyping also avoids the unintuitive conjunctive types introduced by *VariantsK*. For example, the type [< `A of bool & int | `B] → int * bool of page 12 and its more intuitive counterpart [< `B ] → int * bool are translated into equivalent types because of the equivalences $\texttt{`A}(\texttt{bool} \wedge \texttt{int}) \simeq \texttt{`A}(\mathbb{0}) \simeq \mathbb{0}$.

VariantsS and the ℂDuce calculus

VariantsS differs in several respects from the system in Castagna *et al.* (2014) and Castagna, Nguyễn, Xu, and Abate (2015) – which we refer to as ℂDuce since it is the core of the polymorphic version of the language. We impose two main restrictions to simplify the system; there are also a few differences introduced to bring the presentation closer to *VariantsK*.

TYPE-CASE ON ARROW TYPES ℂDuce allows us to discriminate between different arrow types with a type-case construct. In the full language, this is integrated in pattern matching. For instance,

$$\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ (\mathsf{int} \to \mathsf{int}) \to x\ 0 \mid \_ \to 0$$

if applied to a function f from integers to integers, yields the result of f applied to 0; if applied to anything else, it yields 0.

This feature does not seem to be of much practical use; Castagna, Nguyễn, Xu, and Abate (2015) report it has never been used in their programming experience with ℂDuce. Probably, omitting it altogether or only allowing patterns to distinguish functions from non-functions (and not functions of different types from each other) would not be restrictive in concrete use. It has been included for continuity with monomorphic ℂDuce. However, in the polymorphic extension it results in a more complex system, as regards both typing and implementation.

In particular, the complexities arise because the type of a polymorphic function changes if it is instantiated to apply it to some value. For example, a function of type $\alpha \to (\alpha \to \alpha)$, applied to a value of type int, does not yield a function of type $\alpha \to \alpha$: the polymorphic type is instantiated to apply it, so the result has type $\mathsf{int} \to \mathsf{int}$. On the one hand, this means we must keep information on the type of polymorphic functions and update it at run-time with the appropriate instantiation (this is a source of complexity, though the implementation can be made efficient in practice). On the other, the semantics is theoretically non-deterministic, and in practice will be implementation-dependent, if the instantiation of polymorphic functions is inferred. For example, if a function of type $\alpha \to (\alpha \to \alpha)$ is applied to 2, the result might have type $\mathsf{int} \to \mathsf{int}$ or $2 \to 2$ (or yet another) depending on which instantiation is inferred. Neither of these types is a subtype of the other, so there might not be a meaningful canonical choice.

We remove this feature since it is not present in OCaml. Only catch-all patterns accepts functions, so they may not be distinguished among themselves or from other values by pattern matching. Thus, the type system is simplified and this issue disappears; the implementation can discard types at run-time. While our choice depends on the practical goal of our study, the greater simplicity and the removal of this source of non-determinism would make this a reasonable choice also to design a new language with set-theoretic types.

For a new language, an intermediate solution might be adding patterns for non-arrow types and a special pattern which accepts all functions and no other value. This does not complicate the system and it can prove useful in practice. For instance, a pretty-printing function like the one in the OCaml top-level interpreter might use it to render functions as <fun> while it prints other values according to their types. In their work on typing untyped languages, Tobin-Hochstadt and Felleisen (2008) include a procedure? predicate to distinguish functions from other values; this is an indication that such a pattern might be in use in untyped programming.

OVERLOADED FUNCTIONS The second main difference is that we only allow abstractions to be typed with a single arrow type and not with an intersection of them. This is a major restriction, in that it means we cannot type overloaded functions precisely and thus we lose precision in the typing of pattern matching. For instance, we cannot type

$$\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ \text{`}\mathsf{A}(\_) \to \mathsf{true} \mid \text{`}\mathsf{B}(\_) \to \mathsf{false}$$

with the type

$$(\text{`}\mathsf{A}(\mathbb{1}) \to \mathsf{true}) \wedge (\text{`}\mathsf{B}(\mathbb{1}) \to \mathsf{false})$$

which would describe the function exactly. Rather, we must choose the less precise type

$$(\text{`}\mathsf{A}(\mathbb{1}) \vee \text{`}\mathsf{B}(\mathbb{1})) \to \mathsf{bool}$$

which is a supertype. We have imposed this restriction to have a less powerful system for which to study reconstruction (which would be undecidable otherwise). We discuss in Section 6.1 the extension of *VariantsS* which allows overloaded functions.

OTHER DIFFERENCES As for the differences in presentation, for concreteness we directly study the system with many-branch pattern matching and let-polymorphism. Conversely, the CDuce calculus is described with type-case and without let; both aspects are discussed afterwards and the typing of pattern matching is presented in the appendices.

We use type schemes to handle let-polymorphism, for consistency with ML. In CDuce, conversely, a set Δ of type variables is used to record the variables which cannot be instantiated. This Δ is an argument of the typing judgment.

This is not just a cosmetic change, as the system in CDuce allows the instantiation of fewer type variables than ours. However, the difference is immaterial in practice: it only concerns variables which appear in the codomain of an arrow type but not anywhere else, and these can be replaced by $\mathbb{0}$ in practice. In CDuce, while typing the body of an abstraction of type $t_1 \to t_2$, all variables occurring in t_1 and t_2 are considered monomorphic and may not be instantiated. In *VariantsS*, the variables in t_1 become monomorphic

because they are added to the environment, but this is not the case for those in t_2. Our system is sound because, if a variable $\beta \in var(t_2)$ does not occur in the environment nor in t_1, the only expressions that have type β are those that have type $\mathbb{0}$ and thus, by subsumption, they also have any other type; it is then sound to instantiate β with any other type.

5 *Reconstruction for set-theoretic types*

We turn to the study of type reconstruction for the *VariantsS* type system. While reconstruction is undecidable in general in the presence of intersection types, such types have a limited role in our typing rules: abstractions must be typed with a single arrow type and not an intersection. Indeed, we describe here sound and complete reconstruction for a restriction of *VariantsS* where we remove let-polymorphism. While removing let-polymorphism is quite limiting, the typing of variants does not rely on it directly, so this system is still interesting for our study.

Later, we extend reconstruction to encompass let-polymorphism. We keep a less significant restriction: we do not exclude useless branches in the typing of pattern matching, since during reconstruction we do not know yet whether a branch will be useless. This is not very limiting as it only matters if we have redundant patterns. We prove soundness, but not completeness (though we conjecture it holds). There are significant complications in dealing with let-polymorphism in our system as opposed to ML.

Type reconstruction for a type system with set-theoretic types and semantic subtyping has been studied in Castagna, Nguyễn, Xu, and Abate (2015). We reuse their results related to the resolution of the *tallying problem* (the problem of finding substitutions satisfying a set of subtyping constraints), which is also used to define a typing algorithm for the explicitly-typed language. Our contribution is threefold. First, we show that our restriction on the use of intersection types allows us to give a reconstruction system that is complete with respect to the deductive type system. Second, we outline the extension to let-polymorphism, though we do not show completeness; we suggest some possibilities for future work. Third, we describe reconstruction for full pattern matching.

We divide type reconstruction in two phases: the generation of constraints from expressions and the resolution of these constraints. These two phases can be distinguished straightforwardly in the simply-typed λ-calculus, where resolution is unification (Wand, 1987; Pierce, 2002, Chapter 22). With let-polymorphism, reconstruction is often described intertwining the two phases (see, for instance, Leroy, 1992). We keep them separated, following the approach of Pottier and Rémy (2005) and the simpler formulation for the Hindley-Milner type system in Rémy (2013); we describe a simple strategy for constraint resolution. This approach requires us to use a structured form of constraints, which we introduce from the start for consistency.

$$\textit{Ts-Match}' \quad \frac{\Gamma \vdash_s e_0 : t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \qquad \forall i \in I \qquad \Gamma, gen_\Gamma(t_i /\!/ p_i) \vdash_s e_i : t'_i}{\Gamma \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} : \bigvee_{i\in I} t'_i}$$

$$\textit{Ts-MatchM} \quad \frac{\Gamma \vdash_s e_0 : t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \qquad \forall i \in I \qquad \Gamma, t_i /\!/ p_i \vdash_s e_i : t'_i}{\Gamma \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} : \bigvee_{i\in I} t'_i}$$

FIGURE 5.1 Restricted typing rules for pattern-matching expressions.

5.1 RESTRICTION OF THE TYPE SYSTEM

As mentioned above, we restrict *VariantsS* in order to study type reconstruction for it. A minor restriction, which we enforce throughout this chapter, is obtained by replacing the rule *Ts-Match* (in Figure 4.2 on page 40) with the rule *Ts-Match′* in Figure 5.1.

The restricted rule requires every branch to be typed, even if the corresponding t_i is empty. During reconstruction, we do not know t_0 until the end, and we generate constraints by using some variable α in its place. We cannot know whether some t_i will be empty or not, since that depends on the instantiation of α. For instance, if the first pattern is the constant true, having $\alpha = \mathsf{true}$ makes any subsequent pattern redundant, while it is not so if $\alpha = \mathsf{bool}$. We could surely detect some redundancy during reconstruction (repeated patterns, for example), but we do not do so for simplicity.

Note that $t_i /\!/ p_i$ may now produce empty types, which did not happen before because we only generated the pattern environment when $t_i \not\leq \mathbb{0}$. Introducing these empty types in the type environment to type a branch is sound, since it is more restrictive than not typing that branch at all, as *Ts-Match* does (*i.e.* the system is necessarily sound since it types fewer programs than *VariantsS*, which is itself sound).

As mentioned above, this restriction is not so significant as it only makes a difference when there are redundant patterns: if there is no redundancy, the two systems are equivalent. Realistically, a type reconstruction algorithm should signal redundancy with a warning. It does not seem too limiting to also allow it to fail occasionally when it is present.

The other restriction is much stronger. We consider a variant of the type system without let-polymorphism. In it, we assume that type environments only contain types (*i.e.* trivial type schemes), so that type instantiation for variables effectively does not occur. We change the typing rule for pattern matching so it does not perform generalization; the modified rule *Ts-MatchM* is in Figure 5.1. It also includes the first restriction.

5.2 RECONSTRUCTION WITHOUT LET-POLYMORPHISM

In this section, we consider the problem of type reconstruction for *VariantsS* without let-polymorphism. We consider the typing relation $\Gamma \vdash_s e\colon t$ given by the rules in Figure 4.2 on page 40, except *Ts-Match*, plus *Ts-MatchM*. We assume Γ not to contain type schemes with quantified variables.

Type reconstruction for a program (a closed expression) e consists in finding a type t such that $\varnothing \vdash_s e\colon t$ can be derived; we see it as finding a type substitution θ such that $\varnothing \vdash_s e\colon \alpha\theta$ holds. We generalize this formulation to non-closed expressions and to reconstruction where we partially know the type to be reconstructed. Thus we say that type reconstruction consists – given an expression e, a type environment Γ, and a type t – in computing a type substitution θ such that $\Gamma\theta \vdash_s e\colon t\theta$ holds, if any such θ exists.

Reconstruction in our system comprises two phases. In the first, *constraint generation*, we build a set of *constraints* to record the conditions under which an expression e may be given type t. Constraints are built from e and t alone, without using Γ: the free variables of e appear directly in the constraints rather than being replaced by their types.

In the second phase, *constraint solving*, we compute the substitution. We break solving into two steps. *Constraint rewriting* converts our constraints to a simpler form, that we call a *type-constraint set*: a set of subtyping constraints between types. In doing so, we replace expression variables with their types, looking them up in Γ. Finally, *type-constraint solving* computes the solution to the type-constraint set, using the existing algorithm to solve the *tallying problem* described in Castagna, Nguyễn, Xu, and Abate (2015).

The decision to use constraints with expression variables and then convert them to a simpler form is irrelevant at this point – we might just use the simpler form, since rewriting is uninteresting. We introduce the two forms because they allow us to keep constraint generation and solving separate also in the presence of let-polymorphism. This may make the system clearer and aid in studying flexible strategies for resolution; we do not take much advantage of it currently since we reuse the existing algorithm for tallying, but it should be more suitable for future work. For consistency, we use both forms from the beginning.

Constraint generation

We begin by defining the language of constraints we will consider. Constraint generation yields a set of constraints of the form defined below.

DEFINITION 5.1: Constraints A *constraint* c is a term inductively generated by the following grammar:

$$c ::= t \dot{\leq} t \mid x \dot{\leq} t \mid \mathsf{def}\ \Gamma\ \mathsf{in}\ \{c, \ldots, c\}$$

$$\textit{TRs-Var}\ \frac{}{x\colon t \Rightarrow \{x \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Const}\ \frac{}{c\colon t \Rightarrow \{c \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-Abstr}\ \frac{e\colon \beta \Rightarrow C}{\lambda x.e\colon t \Rightarrow \{\mathsf{def}\ \{x\colon \alpha\}\ \mathsf{in}\ C, \alpha \to \beta \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Appl}\ \frac{e_1\colon \alpha \to \beta \Rightarrow C_1 \qquad e_2\colon \alpha \Rightarrow C_2}{e_1\ e_2\colon t \Rightarrow C_1 \cup C_2 \cup \{\beta \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-Pair}\ \frac{e_1\colon \alpha_1 \Rightarrow C_1 \qquad e_2\colon \alpha_2 \Rightarrow C_2}{(e_1, e_2)\colon t \Rightarrow C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Tag}\ \frac{e\colon \alpha \Rightarrow C}{\grave{}tag(e)\colon t \Rightarrow C \cup \{\grave{}tag(\alpha) \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-MatchM}\ \frac{\begin{array}{c} e_0\colon \alpha \Rightarrow C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \forall\, i \in I \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow C'_i \\ C = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\, \mathsf{def}\ \Gamma_i\ \mathsf{in}\ C'_i \mid i \in I \,\} \cup \{\alpha \mathrel{\dot{\leq}} \bigvee_{i\in I} \lceil p_i \rfloor, \beta \mathrel{\dot{\leq}} t\} \end{array}}{\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow C}$$

FIGURE 5.2 Constraint generation rules (without let-polymorphism).

where, in constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in}\ \{c_1, \ldots, c_n\}$, the range of Γ only contains types.

A *constraint set* C is a finite set of constraints. □

A constraint of the first form, $t \mathrel{\dot{\leq}} t'$, requires $t\theta \leq t'\theta$ to hold for the final substitution θ. One of the form $x \mathrel{\dot{\leq}} t$ constrains the type of x in the same way. A definition constraint $\mathsf{def}\ \Gamma\ \mathsf{in}\ C$ introduces new expression variables, as we do in abstractions and pattern matching; these variables may then occur in the constraint set quantified by the definition. We use these constraints to introduce monomorphic bindings (environments with types and not type schemes), while, in the extension to let-polymorphism, we will add a different form for bindings we intend to generalize.

We now define constraint generation. For pattern-matching expressions, we use an auxiliary relation to generate an environment and its constraints for each pattern.

DEFINITION 5.2: Constraint generation The constraint generation relation $e\colon t \Rightarrow C$ is defined by the rules in Figure 5.2. We assume all variables introduced by the rules to be fresh. □

DEFINITION 5.3: Environment generation for pattern matching The environment generation relation for pattern matching $t /\!/\!/ p \Rightarrow (\Gamma, C)$ is defined by the rules in Figure 5.3. We assume all variables introduced by the rules to be fresh. □

We impose a freshness condition on the variables informally, assuming all variables introduced by the rules to be distinct from each other and from

$$\frac{}{t /\!/\!/ \_ \Rightarrow (\varnothing, \varnothing)} \qquad \frac{}{t /\!/\!/ x \Rightarrow (\{x\colon t\}, \varnothing)} \qquad \frac{}{t /\!/\!/ c \Rightarrow (\varnothing, \varnothing)}$$

$$\frac{\alpha_1 /\!/\!/ p_1 \Rightarrow (\Gamma_1, C_1) \qquad \alpha_2 /\!/\!/ p_2 \Rightarrow (\Gamma_2, C_2)}{t /\!/\!/ (p_1, p_2) \Rightarrow (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\})} \qquad \frac{\alpha /\!/\!/ p \Rightarrow (\Gamma, C)}{t /\!/\!/ \grave{t}ag(p) \Rightarrow (\Gamma, C \cup \{t \dot{\leq} \grave{t}ag(\alpha)\})}$$

$$\frac{t /\!/\!/ p_1 \Rightarrow (\Gamma_1, C_1) \qquad t /\!/\!/ p_2 \Rightarrow (\Gamma_2, C_2)}{t /\!/\!/ p_1 \& p_2 \Rightarrow (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2)}$$

$$\frac{(t \wedge \lbag p_1 \rbag) /\!/\!/ p_1 \Rightarrow (\Gamma_1, C_1) \qquad (t \setminus \lbag p_1 \rbag) /\!/\!/ p_2 \Rightarrow (\Gamma_2, C_2)}{t /\!/\!/ p_1 | p_2 \Rightarrow (\{\, x\colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathit{capt}(p_1) \,\}, C_1 \cup C_2)}$$

FIGURE 5.3 Constraint generation for pattern environments.

variables that already appear in t. We also assume they do not appear in the environment Γ we will use for rewriting. In the Appendix, we give another formulation of these relations which keeps track explicitly of all variables we introduce, to use when proving completeness.

Constraint generation for variables and constants just yields a subtyping constraint on the variable or the constant type. For an abstraction $\lambda x.e$, we generate constraints for the body and wrap them into a definition constraint which binds x to a fresh type variable α; we also require the type $\alpha \to \beta$ we are reconstructing to be a subtype of t. The rules for applications, pairs, and tags are similar: we introduce fresh variables as types for the sub-expressions and add a subtyping constraint. The rule for pattern-matching expressions is more complicated. We construct an environment (together with a set of constraints) for each pattern. We generate constraints for the expression to be matched and for each branch; those for the branches are wrapped in definition constraints which introduce the new environments. We also add the constraint $\alpha \leq \bigvee_{i \in I} \lbag p_i \rbag$ for exhaustiveness.

Note that each rule includes a constraint to require the type we reconstruct to be a subtype of t. We do so for simplicity since we aim to prove completeness with respect to a system with subsumption. By changing the definition of completeness we give later, we might avoid the constraints in the cases of application and pattern matching, as subsumption can be pushed to the sub-expressions. In practice, the constraint $\beta \dot{\leq} t$ can be dropped from the rules *TRs-Appl* and *TRs-MatchM* in an implementation.

The relation $t /\!/\!/ p \Rightarrow (\Gamma, C)$ builds an environment together with a set of constraints to relate the new variables it introduces with t. The rules are mostly straightforward. We need to introduce new variables when patterns extract types below type constructors; this parallels the use of projection operators in environment generation for the deductive system. For union patterns, we just build the pointwise union of the two environments.

$$\frac{\forall i \in I \qquad \Gamma \vdash c_i \rightsquigarrow D_i}{\Gamma \vdash \{\, c_i \mid i \in I \,\} \rightsquigarrow \bigcup_{i \in I} D_i} \qquad \frac{}{\Gamma \vdash t \mathrel{\dot{\leq}} t' \rightsquigarrow \{t \mathrel{\dot{\leq}} t'\}}$$

$$\frac{}{\Gamma \vdash x \mathrel{\dot{\leq}} t \rightsquigarrow \{\Gamma(x) \mathrel{\dot{\leq}} t\}} \qquad \frac{\Gamma, \Gamma' \vdash C \rightsquigarrow D}{\Gamma \vdash \mathsf{def}\ \Gamma'\ \mathsf{in}\ C \rightsquigarrow D}$$

FIGURE 5.4 Constraint rewriting rules (without let-polymorphism).

Constraint rewriting

The first step of constraint solving consists in rewriting constraint sets into a simpler form. In this system, rewriting is trivial as we only substitute expression variables with their types. In the extension to let-polymorphism, this phase will actually perform multiple steps of rewriting and resolution, one for each pattern-matching expression.

The simpler form of constraints only includes subtyping constraints.

DEFINITION 5.4: Type-constraint set A *type-constraint set* D is a set of constraints of the form $t \mathrel{\dot{\leq}} t'$, where t and t' are types.

We say that a type substitution θ *satisfies* a type-constraint set D, and we write $\theta \Vdash D$, if $t\theta \leq t'\theta$ holds for each constraint $t \mathrel{\dot{\leq}} t'$ in D. □

We define constraint rewriting, which operates in a type environment and produces a type-constraint set from a constraint set.

DEFINITION 5.5: Constraint rewriting The relation $\Gamma \vdash c \rightsquigarrow D$ between type environments, constraints or constraint sets, and type-constraints is defined by the rules in Figure 5.4. □

The relation is actually a function of the environment and the constraints (let-polymorphism will introduce non-determinism). A subtyping constraint is rewritten to itself. A variable type constraint $x \mathrel{\dot{\leq}} t$ is turned into a subtyping constraint by replacing x with its type in the environment (there is no let-polymorphism and thus no need for instantiation); rewriting is defined only if x is bound in the environment. An environment definition constraint is rewritten by rewriting the inner constraint set in the expanded environment. Constraint sets are rewritten pointwise.

The output of constraint rewriting is thus a set of subtyping constraints between types, as opposed to a set of equality constraints such as we would have for the simply-typed λ-calculus. Equality constraints can be solved by unification; here, we need a different algorithm.

Type-constraint solving: the tallying problem

Castagna, Nguyễn, Xu, and Abate (2015) define the *tallying problem* as the problem – in our terminology – of finding a substitution that satisfies a certain type-constraint set. (Their definition also restricts which variables the substitution can be defined on, but we do not need this here.)

DEFINITION 5.6 Let D be a type-constraint set. A type substitution θ is a solution to the *tallying problem* of D if it satisfies D, that is, if $\theta \Vdash D$. □

Tallying plays the same role in our system as the unification problem does in ML. It is used in the typing algorithm we have mentioned – where abstractions are annotated with their types – to infer the instantiation of polymorphic functions. Here, we use it to extract a substitution from the constraints generated from an expression.

There is, however, a very significant difference from ML. A set of equality constraints between ML types has a most general solution whenever it can be solved. Conversely, there is no most general solution to a set of subtyping constraints in our system. Rather, we can find a finite set of solutions that are more general than any other if taken together, but each of which is incomparable to the others. This means we cannot find a type scheme that represents exactly all types we can reconstruct for an expression; rather, we need a finite set of type schemes to do so.

This difference stems from our semantic definition of subtyping, in particular as regards the empty type and its interaction with type constructors. We consider here the case of pairs; the same reasoning applies to variant and arrow types. We have remarked before that $\mathbb{0} \times t \simeq \mathbb{0} \simeq t \times \mathbb{0}$ for any t: a product type with an empty component is equivalent to the empty type itself. More generally, $t_1 \times t_2 \simeq \mathbb{0}$ holds if either $t_1 \simeq \mathbb{0}$ or $t_2 \simeq \mathbb{0}$.

This results in subtyping for pairs which behaves unlike that of a straightforward syntactic treatment. In the latter, we might say $t_1 \times t_2 \leq t_1' \times t_2'$ holds if both $t_1 \leq t_1'$ and $t_2 \leq t_2'$ hold. In our case, that is a possibility; however, $t_1 \simeq \mathbb{0}$ and $t_2 \simeq \mathbb{0}$ are two other conditions either of which is enough to satisfy subtyping. Thus, a type-constraint set has no most general solution. For instance, the set $\{\alpha \times \beta \mathrel{\dot{\leq}} \gamma \times \delta\}$ can be solved in three incompatible ways: instantiating α with $\mathbb{0}$, instantiating β with $\mathbb{0}$, or instantiating α and β with subtypes of γ and δ respectively.

We have an algorithm to solve the tallying problem, that is sound and complete. By completeness we mean that it returns a set of substitutions that are the most general, if taken together.

PROPERTY 5.7: Tallying algorithm There exists an algorithm $\mathit{tally}(\cdot)$ such that, for any type-constraint set D, $\mathit{tally}(D)$ is a finite set of type substitutions, possibly empty. □

The algorithm $\mathit{tally}(\cdot)$ is $\mathit{Sol}_\Delta(\cdot)$ defined in Castagna, Nguyễn, Xu, and Abate (2015) – adapted for the presence of variant types – where we always take $\Delta = \varnothing$. Soundness and completeness are proved there.

THEOREM 5.1: Soundness and completeness of $\mathit{tally}(\cdot)$ *Let D be a type-constraint set. For any type substitution θ:*

- *if $\theta \in \mathit{tally}(D)$, then $\theta \Vdash D$;*
- *if $\theta \Vdash D$, then $\exists \theta' \in \mathit{tally}(D), \theta''.\ \forall \alpha \notin \mathit{var}(\theta').\ \alpha\theta \simeq \alpha\theta'\theta''$.*

Hence, given a type-constraint set, we can use $\mathit{tally}(\cdot)$ to either find a number of solutions or determine it has no solution: $\mathit{tally}(D) = \varnothing$ occurs if and only if there exists no θ such that $\theta \Vdash D$.

The algorithm produces substitutions whose domain is a subset of the set of variables in the type-constraint set. The substitutions only introduce fresh variables (*i.e.* we can pick the variables they introduce so as not to overlap with other sets). The solutions are all idempotent substitutions because they replace all variables they are defined on with types that only contain fresh variables (on which they are not defined).

Properties of type reconstruction

Our type reconstruction system is both sound and complete with respect to the type system without let-polymorphism we are considering in this section. We state these properties in terms of constraint rewriting.

THEOREM 5.2: Soundness of constraint generation and rewriting *Let e be an expression, t a type, and Γ a type environment. If $e\colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_s e\colon t\theta$.*

THEOREM 5.3: Completeness of constraint generation and rewriting *Let e be an expression, t a type, and Γ a type environment. Let θ be a type substitution such that $\Gamma\theta \vdash_s e\colon t\theta$.*

Let $e\colon t \Rightarrow C$. There exist a type-constraint set D and a type substitution θ', with $\mathit{dom}(\theta) \cap \mathit{dom}(\theta') = \varnothing$, such that $\Gamma \vdash C \rightsquigarrow D$ and $(\theta \cup \theta') \Vdash D$.

In the Appendix we give a slightly different statement for completeness because we keep track explicitly of the variables introduced by constraint generation and require θ' to be defined only on those variables.

Together, these properties and the properties of tallying imply that type reconstruction is sound and complete. Reconstruction for an expression – if successful – yields a set of substitutions and therefore a set of possible types, each of which is correct.

5.3 ADDING LET-POLYMORPHISM

We discuss an extension of our type reconstruction system that handles the type system with let-polymorphism. In this section, we consider the typing relation $\Gamma \vdash_{\mathrm{s}} e\colon t$ defined by the rules in Figure 4.2 on page 40, except *Ts-Match*, plus *Ts-Match′*.

Let-polymorphism comes into play in two rules of our system. *Ts-Match′* generalizes the types in each environment it generates from the pattern. Hence we will have non-trivial type schemes in the environment, and *Ts-Var* must perform instantiation. We can handle the latter by changing constraint rewriting for $x \mathrel{\dot\leq} t$ constraints: we do not rewrite them to $\Gamma(x) \mathrel{\dot\leq} t$ – which makes no sense if $\Gamma(x) = \forall A.\, t_x$ has quantified variables – but to $t_x\theta \mathrel{\dot\leq} t$, where θ is an instantiation of the variables in A with fresh variables.

Turning to pattern matching, we can first examine the simpler case of let-declarations. In ML, type reconstruction for an expression $\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1$ in some environment Γ can be described in three steps:

- reconstruct the type of e_0 in Γ;
- if reconstruction is successful, let t_0 be the type obtained and generalize it with respect to Γ to obtain a type scheme s_0;
- reconstruct the type of e_1 in the environment $\Gamma, \{x\colon s_0\}$.

The drawback of this approach is that it mixes constraint generation and constraint solving. We follow other constraint-based approaches (in particular Pottier and Rémy, 2005; Rémy, 2013) in continuing to enforce a separation of the two steps – thanks to constraints with expression variables and to a new constraint we introduce now. Besides arguably making the system clearer, this separation allows one to study and compare different strategies for solving without changing the form of constraints themselves. Here, we formalize the strategy described above, which allows us to reuse the tallying algorithm as much as possible.

The lack of a most general solution to the tallying problem now poses difficulties that do not exist in ML: in the strategy above, we will not reconstruct a single type t_0 for e_0, but rather a finite number of different types. In this formulation, we describe constraint rewriting as a non-deterministic algorithm which may pick any of the solutions to proceed.

We prove soundness for this system, that is, we prove that any choice of solution leading to successful reconstruction yields a valid type. We do not prove completeness, which in this context would mean that there is always a choice of solution which will lead to successful reconstruction at the end (if the expression is well-typed). At the end of this section, we discuss possible approaches to solve the complication caused by the lack of a general solution.

Constraint generation

We introduce a form of constraint for let-polymorphic constructs. In the case of let, it should record the constraints generated by e_0 and e_1, separately, and the type we will introduce before solving the constraints for e_1 (actually a type variable, which will be instantiated by the solution we get for e_0). For instance, $\mathsf{let}\ x = 3\ \mathsf{in}\ (x, \mathsf{true})$ generates a constraint

$$\{\mathsf{let}\,[\ \underbrace{\{3 \dot{\leq} \alpha\}}_{\text{constraints for } e_0}\ ]\ \underbrace{\{x\colon \alpha\}}_{\text{new binding}}\ \mathsf{in}\ \underbrace{\{\alpha_1 \times \alpha_2 \dot{\leq} \beta, x \dot{\leq} \alpha_1, \mathsf{true} \dot{\leq} \alpha_2\}}_{\text{constraints for } e_1}, \beta \dot{\leq} t\}$$

where t is the type we want to reconstruct.

Since match expressions have many branches and different environments for each, we define let constraints in a more general form. We add this new production to the grammar of constraints presented in Definition 5.1:

$$c ::= \cdots \mid \mathsf{let}\,[C](\Gamma_i\ \mathsf{in}\ C_i)_{i\in I}$$

where the range of every type environment Γ_i in a constraint of the form $\mathsf{let}\,[C_0](\Gamma_i\ \mathsf{in}\ C_i)_{i\in I}$ only contains types.

We change constraint generation by replacing the rule *TRs-MatchM* with the new rule *TRs-Match* in Figure 5.5; all other rules are unchanged. This general rule for pattern matching can be simplified to

$$\frac{e_0\colon \alpha \Rightarrow C_0 \qquad e_1\colon \beta \Rightarrow C_1}{\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\colon t \Rightarrow \{\mathsf{let}\,[C_0]\{x\colon \alpha\}\ \mathsf{in}\ C_1, \beta \dot{\leq} t\}}$$

in the special case of let-declarations.

Constraint solving

We extend constraint rewriting to handle let constraints. In doing so, we mix rewriting and type-constraint solving: to rewrite $\mathsf{let}\,[C_0](\Gamma_i\ \mathsf{in}\ C_i)_{i\in I}$, we first rewrite C_0 to a type-constraint set D_0, then we compute $\mathit{tally}(D_0)$. If there is no solution, rewriting fails. If we find at least one solution, we choose one of them non-deterministically and proceed. Let θ_0 be the chosen solution; we rewrite each C_i in an environment extended by adding $\Gamma_i\theta_0$, generalized with respect to the environment $\Gamma\theta_0$.

To have soundness we must ensure that every substitution θ we get as our final result (*i.e.* by solving the type-constraint set we get by rewriting the constraints of the whole program) can be obtained as a special case of θ_0. Constraint rewriting for let therefore adds constraints to ensure this will happen. We define below the generation, from a type substitution θ_0, of a type-constraint set $\mathit{equiv}(\theta_0)$ such that every substitution θ which solves $\mathit{equiv}(\theta_0)$ will be equivalent to $\theta \circ \theta_0$.

$$\textit{TRs-Match}\ \frac{\begin{array}{c} e_0\colon \alpha \Rightarrow C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \forall\, i \in I \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow C_i' \\ C_0' = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\alpha \mathrel{\dot{\leq}} \bigvee_{i\in I} \lceil p_i \rfloor\} \end{array}}{\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow \{\mathsf{let}\,[C_0'](\Gamma_i\ \mathsf{in}\ C_i')_{i\in I}, \beta \mathrel{\dot{\leq}} t\}}$$

FIGURE 5.5 Constraint generation rule for let-polymorphic match.

$$\frac{\forall\, i \in I \qquad \Gamma \vdash c_i \rightsquigarrow D_i}{\Gamma \vdash \{\, c_i \mid i \in I \,\} \rightsquigarrow \bigcup_{i\in I} D_i} \qquad \frac{}{\Gamma \vdash t \mathrel{\dot{\leq}} t' \rightsquigarrow \{t \mathrel{\dot{\leq}} t'\}} \qquad \frac{\Gamma(x) = \forall\{\alpha_1, \ldots, \alpha_n\}.\, t_x}{\Gamma \vdash x \mathrel{\dot{\leq}} t \rightsquigarrow \{t_x[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \mathrel{\dot{\leq}} t\}}$$

$$\frac{\Gamma, \Gamma' \vdash C \rightsquigarrow D}{\Gamma \vdash \mathsf{def}\ \Gamma'\ \mathsf{in}\ C \rightsquigarrow D} \qquad \frac{\begin{array}{c} \Gamma \vdash C_0 \rightsquigarrow D_0 \qquad \theta_0 \in \mathit{tally}(D_0) \\ \forall\, i \in I \qquad \Gamma, \mathit{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i \rightsquigarrow D_i \end{array}}{\Gamma \vdash \mathsf{let}\,[C_0](\Gamma_i\ \mathsf{in}\ C_i)_{i\in I} \rightsquigarrow \mathit{equiv}(\theta_0) \cup \bigcup_{i\in I} D_i}$$

FIGURE 5.6 Constraint rewriting rules (with let-polymorphism).

DEFINITION 5.8: Equivalent type-constraint set Given a type substitution θ, we define its *equivalent type-constraint set* $\mathit{equiv}(\theta)$ as

$$\mathit{equiv}(\theta) = \bigcup_{\alpha\in \mathit{dom}(\theta)} \{\alpha \mathrel{\dot{\leq}} \alpha\theta,\ \alpha\theta \mathrel{\dot{\leq}} \alpha\}\,. \qquad \square$$

The constraint rewriting relation $\Gamma \vdash c \rightsquigarrow D$ for the system with let-polymorphism is defined by the rules in Figure 5.6. As compared to rewriting without let-polymorphism, we change the rule for variable constraints and add the new rule for let constraints. For a constraint $x \mathrel{\dot{\leq}} t$, we build an instance of the type scheme $\Gamma(x)$ by instantiating its quantified variables with $\{\beta_1, \ldots, \beta_n\}$, which we assume to be fresh. For let constraints we proceed as described above. The type-constraint set we produce contains $\mathit{equiv}(\theta_0)$: this ensures that each solution θ to that type-constraint set is such that $\alpha\theta \simeq \alpha\theta_0\theta$ for every α, meaning it is an extension of θ_0.

The system we have described is sound, as stated by the following result.

THEOREM 5.4: Soundness of constraint generation and rewriting with let-polymorphism *Let e be an expression, t a type, and Γ a type environment. If $e\colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_{\mathrm{s}} e\colon t\theta$.*

Note that, since $\Gamma \vdash C \rightsquigarrow D$ is non-deterministic (we can choose any of the solutions produced by tallying when we solve a let constraint), soundness means that no choice of solution may lead to reconstructing invalid types.

Challenges and future work

The difficulty we have in defining type reconstruction for our system is due to the lack of most general solutions to the tallying problem. Our reconstruction system chooses one of the solutions non-deterministically to proceed. We have proved that it is sound, but not that it is complete. In this setting, completeness would mean that – if an expression is well-typed – there exists a choice of solutions which will lead to successful reconstruction.

We conjecture completeness holds for this system (or some slight modification of it). Having to choose one of the solutions (as opposed to keeping all of them somehow) is not restrictive with respect to our typing rules: alternative solutions correspond to different type schemes, so the typing rules also force us to choose a single one. Indeed, we add the constraints generated by $\mathit{equiv}(\cdot)$ during rewriting precisely to force reconstruction to abide by the choice of solution that has been made.

However, even if the system were complete in this sense, it would still need to explore all possible choices of solutions. As a matter of fact, some solutions are redundant and can be discarded because we can obtain them from the others by subtyping. For instance, if tallying generates for a function both some type $t_1 \to t_2$ (possibly the type we would expect in ML) and the type $\mathbb{0} \to \mathbb{0}$, the latter can be obtained from the former by subsumption. Essentially, $\mathbb{0} \to \mathbb{0}$ states that the application of the function to a diverging expression will diverge. It is a supertype of every type of the form $t_1 \to t_2$, since it is the top type of arrows.[1]

It remains to be seen in which cases we produce multiple solutions that are really significant. In any event, multiple solutions should only arise because of the behaviour of the empty type with respect to subtyping. It should be possible to define a canonical choice of solution to tallying – choosing that which avoids empty types – which is not strictly speaking the 'most general' but should be more useful in practice: the other solutions would only be needed for some situations where we have diverging expressions. By fixing a canonical solution and discarding the others, we might lose completeness with respect to our system, but we should keep it with respect to ML.

An alternative to defining such a canonical choice could be restricting let-polymorphism. Vytiniotis, Peyton Jones, Schrijvers, and Sulzmann (2011) propose a system where top-level let-bindings are generalized, but local ones are not, unless there is an explicit polymorphic type annotation. Their approach recovers principal types in systems where they would not exist with implicit let-generalization (for example, for GADTs), at – they argue – a small practical cost for programmers.

1 Every type of the form $\mathbb{0} \to t$ is equivalent to the top type of arrows.

6 *Extensions and variations*

In this chapter, we outline three ways to extend or modify the *VariantsS* type system described in Chapter 4. In the first extension, we allow the use of intersection types to type abstractions; thanks to this, we can define overloaded functions and type them precisely. In the second, we improve the typing of pattern matching: we use the patterns to refine the types of variables in the expression we match. Finally, the third is a restriction of the type system. The full system is not directly applicable to a type-erasing language like OCaml; hence, we study a semantics which reflects the actual implementation of OCaml and describe how to modify *VariantsS* to make it sound with respect to that semantics.

We study these variations only in the context of the deductive type system and do not study type reconstruction. For the last two, modifying reconstruction as well should be straightforward. Conversely, reconstruction in the presence of functions typed with intersection types is undecidable in general; we would need the system to take into account the programmer's explicit type annotations.

6.1 OVERLOADED FUNCTIONS

In the calculus for polymorphic ℂDuce, we can use intersection types to type abstractions; conversely, our system allows us to derive intersections only by subsumption. This means we cannot express the fact that a function is overloaded in its type. For instance, we can type

$$\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{true} \to \mathsf{false} \mid \mathsf{false} \to \mathsf{true}$$

as $\mathsf{bool} \to \mathsf{bool}$, but not as $(\mathsf{true} \to \mathsf{false}) \wedge (\mathsf{false} \to \mathsf{true})$, which would be more precise. Functions defined by pattern matching can be typed more precisely with intersection types: we can give a different type for each branch.

To add this possibility we consider an explicitly-typed system (as done by Castagna *et al.*, 2014, and Castagna, Nguyễn, Xu, and Abate, 2015): we assume all abstractions to be annotated with their type. Annotated abstractions are treated as un-annotated ones in the semantics.

We define typing in this extension as a relation $\Gamma \vdash_{\mathrm{so}} e\colon t$, given by the rules defining $\Gamma \vdash_{\mathrm{s}} e\colon t$ (in Figure 4.2 on page 40), except *Ts-Var* and *Ts-Abstr*, plus the rules in Figure 6.1.

$$\textit{Tso-Var}\ \frac{\forall i \in I.\ \ t_i \in \mathit{inst}(\Gamma(x))}{\Gamma \vdash_{\text{so}} x\colon \bigwedge_{i\in I} t_i} \qquad \textit{Tso-Abstr}\ \frac{\forall i \in I.\ \ \Gamma, \{x\colon t'_i\} \vdash_{\text{so}} e\colon t_i}{\Gamma \vdash_{\text{so}} \lambda^{\bigwedge_{i\in I} t'_i \to t_i} x.e\colon \bigwedge_{i\in I} t'_i \to t_i}$$

$$\textit{Tso-Inst}\ \frac{\Gamma \vdash_{\text{so}} e\colon t \qquad \forall i \in I.\ \ t_i \in \mathit{inst}(\mathit{gen}_\Gamma(t))}{\Gamma \vdash_{\text{so}} e\colon \bigwedge_{i\in I} t_i}$$

FIGURE 6.1 Modified typing rules for overloaded abstractions.

The three new rules fulfil two different purposes. The rule *Tso-Abstr* allows us to type an abstraction $\lambda x.e$ with an intersection of arrow types, as long as it is explicitly annotated with that intersection and we can derive each of those types. Thus, we type the body e once for each arrow, each time with a different assumption for the type of x. This introduces ad-hoc polymorphism in an explicitly-typed way.

The rule *Tso-Var* plays a different role. It states that a variable can be typed with an intersection of multiple instantiations rather than with a single one. In practice, this means we can treat a parametrically-polymorphic function as if it were overloaded. For example, we can instantiate the type scheme $\forall\alpha.\,\alpha \to \alpha$ into $(\mathsf{bool} \to \mathsf{bool}) \wedge (\mathsf{int} \to \mathsf{int})$, a type which expresses ad-hoc polymorphism.

Finally, the third rule is necessary because we use explicit annotations. In *VariantsS*, instantiation is only used in *Tso-Var*. Now we need it for abstractions as well: we need it, for instance, to derive the type $\mathsf{bool} \to \mathsf{bool}$ for the function $\lambda^{\alpha\to\alpha}x.x$. For clarity, we use a separate rule and do not combine it with *Tso-Abstr*. We keep instantiation in *Tso-Var* – though it is made redundant by *Tso-Inst* – so typing still derives types and not type schemes.

Tso-Abstr influences the typing of pattern matching, though *Ts-Match* is unchanged, because we now type the body of overloaded abstractions multiple times under different assumptions. Consider the negation function at the beginning of this section. Its body will be typed twice, assuming $\{x\colon \mathsf{true}\}$ the first time and $\{x\colon \mathsf{false}\}$ the second; each time, one branch will never be selected, so we exclude it from the output type. In *VariantsS*, branches with $t_i \leq \mathbb{0}$ are useless, here they are not: they are never selected under the assumptions we are considering, but they may be under different ones. A branch is redundant only if it has $t_i \leq \mathbb{0}$ every time we type it.

Algorithmic typing

We can derive a typing algorithm for this system as described in Castagna *et al.* (2014) and Castagna, Nguyễn, Xu, and Abate (2015). However, there

is a significant difference with respect to *VariantsS*: the algorithmic system is not shown to be complete. The difficulty is that we can instantiate type schemes multiple times and take the intersection. Consider the application

$$(\lambda^{(\mathsf{bool}\to\mathsf{bool})\wedge(\mathsf{int}\to\mathsf{int})} i.\ (i\ \mathsf{true}, i\ 1))\ (\lambda^{\alpha\to\alpha} x.x)\ .$$

The typing algorithm must compute whether this application can be made well-typed by instantiating the generalization of $\alpha \to \alpha$ somehow. No single instantiation will work; however, we can intersect two instantiations to get $(\mathsf{bool} \to \mathsf{bool}) \wedge (\mathsf{int} \to \mathsf{int})$, which makes the application well-typed. Another application might require more than two instantiations; there is no upper bound on how many we can take, and it is an open problem whether we can predict how many will be required.

Thus, the algorithm proposed for ℂDuce continues to increase the number of substitution instances it intersects, using heuristics to decide when to stop if no solution is found. Though completeness is not proven, the heuristics seem to be sufficient for practical programming.

Note that this problem concerns the rules for variables and instantiations, not the possibility of having explicitly-overloaded abstractions. The latter might also be used in a system which does not have the former: we would have both parametric and ad-hoc polymorphism, but we would not be able to instantiate parametrically-polymorphic functions into ad-hoc-polymorphic ones.

6.2 Refining the type of a matched expression

When we type a pattern-matching expression in *VariantsS*, we compute precise types for the capture variables of each pattern by considering the pattern itself and all preceding ones. In the same way, we can also refine the types of variables which appear in the expression we are matching.

Consider the map function in OCaml defined as

```
let rec map f ls = match ls with
  | [] → ls
  | hd :: tl → f hd :: map f tl
▸ (α → α) → α list → α list
```

which is semantically equivalent to the usual way of writing it (returning `[]` rather than `ls` from the first branch). Since we return `ls` – of type α `list` – from a branch, `f` is constrained to have type $\alpha \to \alpha$ rather than the more general type $\alpha \to \beta$. However, we only return `ls` if it is `[]`, and then it has type α `list` but also β `list` for any β. By considering this during typing, we are able to derive the general type for this version of `map` as well.

Situations like this occur often while programming in ℂDuce – as well as in OCaml – where we use pattern matching to perform general type-cases. For instance, we might take x, which is only known to be of type $\mathbb{1}$, and match

it against a pattern like bool (true|false in *VariantsS*): in the corresponding branch, we should be able to use x with type bool and not just $\mathbb{1}$. But unless pattern matching implements a strategy such as that described here, the type of x will not be refined in the branch unless we rebind it explicitly, by changing the pattern to bool&x.

A solution to this issue is described in Castagna *et al.* (2014, Appendix E), only for type-case on type variables. We outline here an extension to general pattern matching, not only on variables but also on pairs or variants containing them.[1]

When the matched expression is a variable, the solution is quite simple: we use the type $t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor$ for the variable when it appears in the branch (with let-polymorphism, we also need to generalize it). It is more precise than its type in the environment: it is exactly the type of the values which will match the pattern.

The extension to pairs and variants seems relatively straightforward too: for instance, if (x, y) matches $(\mathsf{true}, 1|2)$, we know that x has type true and y has type $1 \vee 2$. We define this by translating the matched expression to a pattern and then reusing the relation $\cdot /\!/ \cdot$ for pattern environment generation.

A complication arises when the matched expression has repeated variables; for example, if it is (x, x) – admittedly quite unlikely. In that case, we cannot translate it directly to a pattern because pair patterns may not have repeated variables; hence we introduce a different form of pattern. Furthermore, we intersect the types we obtain for the two occurrences of the variable. If (x, x) matches $(\mathsf{true}, 1|2)$, we know x must have type $\mathsf{true} \wedge (1 \vee 2) \simeq \mathbb{0}$. Incidentally, this tells us the branch will never be selected. Thus, we have also refined our criterion to detect redundancy, though it will seldom be noticeable in practice as it matters only if the matched expression contains repeated variables.

The extension

We modify the typing rule *Ts-Match* to update the environment with the refined types for the variables in the matched expression, before it is updated with types for the capture variables. Before we do so, we define the translation from expressions to patterns we will use in the typing rule.

We introduce a new production to patterns, only for internal use of these typing rules (this new form of pattern should not appear in programs).

$$p ::= \cdots \mid \langle p, p \rangle$$

While pair patterns cannot have repeated variables, a pattern $\langle p_1, p_2 \rangle$ can: we add this form to handle repeated variables in expressions. We give no dynamic semantics to these patterns – as they will not appear in programs –

1 This extension has now been implemented in the development version of CDuce.

$$\textit{Tsm-Match}\ \frac{\Gamma \vdash_s e_0 \colon t_0 \qquad t_0 \le \bigvee_{i\in I} \lceil p_i \rfloor \wedge \lceil (\!| e_0 |\!) \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \quad \forall i \in I \begin{cases} t_i' = \mathbb{0} & \text{if } t_i \le \mathbb{0} \text{ or } \exists t \in \mathit{range}(t_i /\!/ (\!| e_0 |\!)).\ t \le \mathbb{0} \\ \Gamma, \mathit{gen}_\Gamma(t_i /\!/ (\!| e_0 |\!)), \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_s e_i \colon t_i' & \text{otherwise} \end{cases}}{\Gamma \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \colon \bigvee_{i\in I} t_i'}$$

FIGURE 6.2 Modified typing rule for pattern-matching expressions.

but we define their accepted type as

$$\lceil \langle p_1, p_2 \rangle \rfloor = \lceil p_1 \rfloor \times \lceil p_2 \rfloor$$

and environment generation for them as

$$t /\!/ \langle p_1, p_2 \rangle = \pi_1(t) /\!/ p_1 \mathbin{\mathbin{\curlywedge}} \pi_2(t) /\!/ p_2$$

where $\mathbin{\curlywedge}$, defined as

$$(\Gamma \mathbin{\curlywedge} \Gamma')(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma) \setminus \mathit{dom}(\Gamma') \\ \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma') \setminus \mathit{dom}(\Gamma) \\ \Gamma(x) \wedge \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(\Gamma') \end{cases}$$

is the pointwise intersection of type environments.

We define a translation of expressions into patterns. It preserves variables and variants, converts pairs to the new form, and turns everything else into a wildcard.

$$(\!| e |\!) = \begin{cases} x & \text{if } e = x \\ \langle (\!| e_1 |\!), (\!| e_2 |\!) \rangle & \text{if } e = (e_1, e_2) \\ `\mathit{tag}(\!| e' |\!)) & \text{if } e = `\mathit{tag}(e') \\ \_ & \text{otherwise} \end{cases}$$

Thus, variables below abstractions, applications, or pattern-matching constructs are ignored, and we only keep those for which we can extract a type.

The modified typing rule, which replaces *Ts-Match*, is shown in Figure 6.2. The main difference is the addition of the new environment $\mathit{gen}_\Gamma(t_i /\!/ (\!| e_0 |\!))$ to type each branch. We add it before the other; in this way, the capture variables in the patterns take precedence over those in $(\!| e_0 |\!)$.

Another difference is that we exclude more branches because we are more precise in identifying redundancy. We ignore branches where environment generation for $(\!| e_0 |\!)$ produces empty types.

Finally, we require $t_0 \le \lceil (\!| e_0 |\!) \rfloor$ to ensure $t_i /\!/ (\!| e_0 |\!)$ is well-defined. This requirement is not restrictive because any well-typed e can be typed with a subtype of $\lceil (\!| e |\!) \rfloor$.

6.3 APPLICABILITY TO OCAML

Until now we have used *Variants* as our syntax and semantics. However, we intended to describe a type system for polymorphic variants which could be applied to OCaml directly, that is, which might replace the current type system without requiring changes in the rest of the implementation. The semantics of *Variants* does not reflect that of OCaml closely enough for this: indeed *VariantsS* – as we will see – is unsound with respect to the actual implementation of OCaml. We introduce here a restriction of the type system which recovers soundness.

Naturally, our work is still limited to the fragment of OCaml we are considering. To make this type system usable for the whole language, we need to study its interaction with many other advanced features of OCaml, from objects to GADTs. In some cases at least, significant future work might be required.[2] However, with the changes presented in this section, our type system is sound with respect to this fragment of the language.

We have remarked in Section 3.4 that *Variants* differs from OCaml in the behaviour of pattern matching because OCaml uses the same run-time representation for some values of different types (for instance, 1 and true), so they cannot be distinguished at run-time. Thus, a comparison between constants such as 1/true might actually succeed in the implementation, even though the constants are different.

In OCaml, such cases never arise because they are blocked by the type system, so we can observe them only by circumventing it.[3] However, in *VariantsS* we can give more precise types, and we can type programs that were ill-typed before. Now we can observe this behaviour in well-typed programs. For example, a function f defined as

$$\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ \mathsf{true} \to \mathsf{true} \mid \_ \to \mathsf{false}$$

can be given the type $\mathbb{1} \to \mathsf{bool}$ in *VariantsS*. This type is sound with respect to *Variants* (the application of f to any value will reduce to either true or false). However, the semantics of *Variants* differs from that of OCaml, because applying f to 1 results in false in the former and true in the latter.

These discrepancies can also lead to unsoundness. The similar function

$$\lambda x.\ \mathsf{match}\ x\ \mathsf{with}\ (\mathsf{true}, \mathsf{true}) \to \mathsf{true} \mid \_ \to \mathsf{false}$$

can be safely applied to any value in *Variants*, but provokes a crash if it is applied to an integer in OCaml.

The problem is that the matching operation in OCaml is not always well-

2 For example, OCaml allows recursive definitions of cyclic values, such as the infinite list defined by let rec x = 1 :: x. Subtyping in *VariantsS* is based on a model which only considers finite values and is unsound when infinite ones are allowed. The study of models with co-inductive values might give means to solve this issue.

3 The Obj.magic function can be used to perform arbitrary type conversions by reinterpreting the underlying representation, like a cast in C.

$$
\begin{aligned}
v/_I\_ &= [\,] \\
v/_I x &= [v/x] \\
v/_I c &= \begin{cases} [\,] & \text{if } v = c \\ \Omega & \text{if } v \in C,\ b_v = b_c, \text{ and } v \neq c \\ \mho & \text{otherwise} \end{cases} \\
v/_I (p_1, p_2) &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v = (v_1, v_2),\ v_1/_I p_1 = \varsigma_1, \text{ and } v_2/_I p_2 = \varsigma_2 \\ \Omega & \text{if } v = (v_1, v_2),\ \exists i.\ v_i/_I p_i = \Omega, \text{ and } \forall i.\ v_i/_I p_i \neq \mho \\ \mho & \text{otherwise} \end{cases} \\
v/_I \text{`}tag(p_1) &= \begin{cases} \varsigma & \text{if } v = \text{`}tag(v_1) \text{ and } v_1/_I p_1 = \varsigma \\ \Omega & \text{if } v = \text{`}tag(v_1) \text{ and } v_1/_I p_1 = \Omega \text{ or if } v = \text{`}tag_1(v_1) \text{ and } \text{`}tag_1 \neq \text{`}tag \\ \mho & \text{otherwise} \end{cases} \\
v/_I p_1 \& p_2 &= \begin{cases} \varsigma_1 \cup \varsigma_2 & \text{if } v/_I p_1 = \varsigma_1 \text{ and } v/_I p_2 = \varsigma_2 \\ \Omega & \text{if } \exists i.\ v_i/_I p_i = \Omega \text{ and } \forall i.\ v_i/_I p_i \neq \mho \\ \mho & \text{otherwise} \end{cases} \\
v/_I p_1 | p_2 &= \begin{cases} v/_I p_1 & \text{if } v/_I p_1 \neq \Omega \\ v/_I p_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 6.3 Semantics of pattern matching in *VariantsI*.

defined: when we compare constants of different types, constants with pairs, and so on, the outcome is implementation-dependent. *Variants* just models it as ordinary failure. The difference, of course, is that when matching is undefined, it could actually result in success or in a crash; conversely, failure is dealt with by checking the next pattern in the match construct.

We introduce a semantics – an 'implementation' semantics *VariantsI* – to model matching accurately; then, we discuss how to adapt *VariantsS* to it.

Semantics

The semantics of *VariantsI* differs from that of *Variants* only in the definition of the pattern-matching operation. We introduce the possibility of undefined results (written $\mho$), distinct from failure (Ω).

In the semantics, we use the function $b_{(\cdot)}$ which assigns each constant c its basic type b_c.

DEFINITION 6.1: Semantics of pattern matching We write $v/_I p$ for the result of matching a value v against a pattern p. We have either $v/_I p = \varsigma$ –

$$
\begin{aligned}
\langle \_ \rangle &= \mathbb{1} \\
\langle x \rangle &= \mathbb{1} \\
\langle c \rangle &= b_c \\
\langle (p_1, p_2) \rangle &= \langle p_1 \rangle \times \langle p_2 \rangle \\
\langle `tag(p_1) \rangle &= `tag(\langle p_1 \rangle) \vee (\mathbb{1}_{\mathrm{v}} \setminus `tag(\mathbb{1})) \\
\langle p_1 \& p_2 \rangle &= \langle p_1 \rangle \wedge \langle p_2 \rangle \\
\langle p_1 | p_2 \rangle &= \langle p_1 \rangle \vee \langle p_2 \rangle
\end{aligned}
$$

FIGURE 6.4 Compatible type of a pattern.

where ς is a substitution defined on the variables in $capt(p)$ – $v /\!\!/_l p = \Omega$, or $v /\!\!/_l p = \mho$. In the first case, we say that v matches p (or that p accepts v); in the second, we say matching fails; in the third, we say it is undefined.

The definition of $v /\!\!/_l p$ is given in Figure 6.3. □

We use the same reduction rules – defined in Figure 3.1 on page 20 – replacing the $\cdot/\cdot$ operation with $\cdot /\!\!/_l \cdot$. Note that the meaning of the rule

$$\textit{R-Match}\ \frac{v /\!\!/_l p_j = \varsigma \qquad \forall\, i < j.\ v /\!\!/_l p_i = \Omega}{\mathsf{match}\ v\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow e_j \varsigma}\ j \in I$$

is changed significantly. In *Variants*, a pattern-matching construct reduces whenever matching succeeds for one branch at least (it reduces to the first successful branch). In *VariantsI* it reduces only if matching succeeds for a branch and is never undefined for previous branches.

Typing

VariantsK is already sound with respect to *VariantsI*: whenever we have both $K \vdash p\colon \tau \Rightarrow \Gamma$ and $K; \varnothing \vdash_{\kappa} v\colon \tau$, we have $v /\!\!/_l p \neq \mho$ as well. Since we require $K \vdash p_i\colon \tau_0 \Rightarrow \Gamma_i$ to hold for all patterns in a match expression, well-typed programs never produce undefined matching results.

This is not the case for *VariantsS*, where we only require exhaustiveness. Now, we should also require the type of the matched expression to be such that it cannot generate $\mho$ with any pattern. To this end, we define the type of the values each pattern is compatible with, that is, those for which matching is defined.

DEFINITION 6.2: Compatible type of a pattern The *compatible type* $\langle p \rangle$ of a pattern p is defined inductively by the equations in Figure 6.4. □

$$\textit{Tsi-Match}\quad \frac{\Gamma \vdash_{\mathrm{s}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \wedge \bigwedge_{i\in I} \langle p_i \rangle \qquad t_i = \left(t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor\right) \wedge \lceil p_i \rfloor \qquad \forall i \in I \begin{cases} t'_i = \mathbb{0} & \text{if } t_i \leq \mathbb{0} \\ \Gamma, \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_{\mathrm{s}} e_i\colon t'_i & \text{otherwise} \end{cases}}{\Gamma \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \bigvee_{i\in I} t'_i}$$

FIGURE 6.5 Modified typing rule for *VariantsI*.

Like the definition of accepted type, this is equivalent to a semantic definition: having $\langle p \rangle$ be the set of all values v such that $v /\!/ p \neq \mathcal{U}$.

In rule *Ts-Match*, we expressed exhaustiveness as $t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor$: that is, each v of type t_0 must be accepted by one pattern at least. Here, we also require compatibility. For v to be accepted by p_j, it must have type $\lceil p_j \rfloor$ and each type $\langle p_i \rangle$, for $i < j$. We ask that t_0 be a subtype of each $\langle p_i \rangle$.

We modify the typing relation $\Gamma \vdash_{\mathrm{s}} e\colon t$ to make it sound with respect to *VariantsI*: we replace *Ts-Match* with the rule *Tsi-Match* in Figure 6.5.

We have been slightly more restrictive than necessary since we require the input type to be compatible with *all* patterns. Actually, patterns which are useless because any value is already covered by previous cases (*e.g.* any pattern that follows a catch-all pattern) need not be considered. A finer condition such as

$$t_0 \leq \bigvee_{i\in I} \left(\lceil p_i \rfloor \wedge \bigwedge_{j<i} \langle p_j \rangle\right)$$

would only make a difference if there is redundancy, so we have chosen this one for simplicity.

Explicit type tags

This restriction makes it impossible to distinguish types in a union by pattern matching, unless they are discriminated by tags: this is necessary because there is no implicit type tagging in OCaml. Hence, to program in the restricted system we need to use variants to build union types, as we already do in OCaml; nevertheless we still retain much of the flexibility introduced in our system by the use of semantic subtyping. We can also use type connectives to define more precise types, especially to avoid the need to write non-exhaustive pattern matching.

In the extreme, a 'boxing' conversion can be applied to the whole program to wrap every expression with an explicit type tag: everything that is typable in *VariantsS* can be made typable in our restriction by such a conversion. In practice, tags will only be needed in a few situations rather than everywhere.

7 Conclusions

We have investigated the use of set-theoretic type connectives and semantic subtyping to design a type system for languages of the ML family featuring polymorphic variants – a useful feature of OCaml that we attempt to make yet more flexible with this work. Our goal has been to show that such a type system is applicable in this setting and can provide expressive and intuitive typing for variants.

Indeed, the set-theoretic system *VariantsS* we have described is strictly more expressive than the current one used in OCaml, which we have formalized as *VariantsK*. It can type some type-safe programs that *VariantsK* cannot type, and it can give more precise types to programs both can type. In particular, using subtyping rather than structural polymorphism for variants means we do not lose polymorphism when variants are used as function arguments. *VariantsS* also allows pattern matching to be typed much more precisely; moreover, we can define exhaustiveness and non-redundancy checking at the level of types, making the formalism arguably more intuitive.

Besides its direct application to OCaml and other languages of the ML family, this work also continues the ongoing study of languages with set-theoretic types and semantic subtyping, such as ℂDuce. We make three contributions. The first is the study of a simpler setting than that of ℂDuce, without type-case on arrow types (a restriction which seems hardly limiting in practice) and overloaded functions; we show how these restrictions give rise to a far less complex system.

The second contribution is the advancement in the study of type reconstruction with set-theoretic types. We first take a restriction of *VariantsS* without let-polymorphism and give a sound and complete type reconstruction system for it. We then extend the system to deal with let-polymorphism and prove soundness – but not completeness – for the extension. We point out the difficulties and suggest directions for future work.

A third contribution is the refinement of pattern matching we present in Section 6.2, which has been integrated in ℂDuce.

Future work

This work can be further developed in two main directions: the integration of our system in OCaml and a more in-depth study of type reconstruction.

To implement our type system as an alternative to the current system of OCaml, we need to study the interaction between it and all the features of the language we have not studied here. These include, for instance, the object system, the module system, cyclic terms, and GADTs. As an intermediate step along the way to a full implementation, we can work on a prototype limited to the fragment we have studied.

An application of this type system is representing XML trees as polymorphic variants and typing them precisely, as ℂDuce does. Currently, precise typing of XML with semantic subtyping is available in OCaml using OCamlDuce (Frisch, 2006), which is also supported by the Ocsigen web programming framework (Balat, Vouillon, and Yakobowski, 2009). However, OCamlDuce juxtaposes the type systems of OCaml and XDuce without combining them; our system would combine them, and hence would allow writing polymorphic transformations of XML documents.

The integration involves work of theoretical interest as well as of immediate application. We intend to study the relationship between set-theoretic types and typing features such as GADTs, as we have studied here their relationship with structural polymorphism.

Research on type reconstruction for systems with set-theoretic types can be furthered in several directions. Such research is also important to the integration of our system in OCaml, as reconstruction is an essential feature of ML-derived languages. Here, we have done a first step by tailoring the work of Castagna, Nguyễn, Xu, and Abate (2015) to our setting, and we have obtained promising results in our restricted system.

At the end of Section 5.3 we have already discussed some ideas for future work. These mostly concern the problem we have encountered: the lack of most general solutions to the tallying problem. We have allowed reconstruction to choose any of the solutions generated by the tallying algorithm. We intend to study the relationship between these solutions to see if a canonical choice can be defined to avoid having to try out all of them. The goal would be to ensure reconstruction is complete with respect to a restriction of the system that is still expressive enough (hopefully, more than ML). An alternative approach can be restricting the implicit generalization of let-bindings, as suggested by Vytiniotis, Peyton Jones, Schrijvers, and Sulzmann (2011).

We will also work on extending reconstruction to the system with overloaded functions. Full reconstruction is undecidable in this case; hence, we must require the programmer to write explicit type annotations and exploit them during reconstruction.

Additionally, constraint solving might benefit from more flexible strategies. Constraint-based systems such as HM(X) enforce a separation between the generation and the solving of constraints; this simplifies the study and comparison of various solving strategies. We have already followed the approach of Pottier and Rémy (2005) to some extent, but so far we have described a single strategy.

A Proofs

A.1 A CALCULUS FOR ML WITH VARIANTS

LEMMA A.1: Generation for values *Let v be a value. Then:*

- *if $K; \Gamma \vdash_{\kappa} v\colon b$, then $v = c$ for some constant c such that $b_c = b$;*
- *if $K; \Gamma \vdash_{\kappa} v\colon \tau' \to \tau$, then v is of the form $\lambda x.e$ and $K; \Gamma, \{x\colon \tau'\} \vdash_{\kappa} e\colon \tau$;*
- *if $K; \Gamma \vdash_{\kappa} v\colon \tau_1 \times \tau_2$, then v is of the form (v_1, v_2), $K; \Gamma \vdash_{\kappa} v_1\colon \tau_1$, and $K; \Gamma \vdash_{\kappa} v_2\colon \tau_2$;*
- *if $K; \Gamma \vdash_{\kappa} v\colon \alpha$, then v is of the form $\backprime tag(v_1)$, $\alpha :: (L, U, T) \in K$, $\backprime tag \in L$, and $K; \Gamma \vdash_{\kappa} v_1\colon \tau_1$ for the only type τ_1 such that $\backprime tag\colon \tau_1 \in T$.*

Proof The typing rules are syntax-directed, so the last rule applied to type a value is fixed by its form. All these rules derive types of different forms, thus the form of the type assigned to a value determines the last rule used. In each case the premises of the rule entail the consequences above. □

Patterns and environment generation

LEMMA A.2: Correctness of environment generation *Let p be a pattern and v a value such that $v/p = \varsigma$. If $K; \Gamma \vdash_{\kappa} v\colon \tau$ and $K \vdash p\colon \tau \Rightarrow \Gamma'$, then, for all $x \in capt(p)$, $K; \Gamma \vdash_{\kappa} x\varsigma\colon \Gamma'(x)$.*

Proof By induction on the derivation of $K \vdash p\colon \tau \Rightarrow \Gamma'$. We reason by cases on the last applied rule.

Cases TPk-Wildcard and TPk-Const There is nothing to prove since $capt(p) = \varnothing$.

Case TPk-Var We have

$$v/x = [v/x] \qquad K \vdash x\colon \tau \Rightarrow \{x\colon \tau\}$$

and must prove $K; \Gamma \vdash_{\kappa} x[v/x]\colon \{x\colon \tau\}(x)$, which we know by hypothesis.

Case TPk-Pair We have

$$K \vdash (p_1, p_2)\colon \tau_1 \times \tau_2 \Rightarrow \Gamma_1' \cup \Gamma_2' \qquad K \vdash p_1\colon \tau_1 \Rightarrow \Gamma_1' \qquad K \vdash p_2\colon \tau_2 \Rightarrow \Gamma_2' .$$

By Lemma A.1, $K; \Gamma \vdash_{\kappa} v\colon \tau_1 \times \tau_2$ implies $v = (v_1, v_2)$ and $K; \Gamma \vdash_{\kappa} v_i\colon \tau_i$ for both i. Furthermore, $(v_1, v_2)/(p_1, p_2) = \varsigma = \varsigma_1 \cup \varsigma_2$, and $v_i/p_i = \varsigma_i$ for both i. For each capture variable x, we can apply the induction hypothesis to the sub-pattern which contains x and conclude.

Case TPk-Tag We have

$$K \vdash \text{`}tag(p_1) \colon \alpha \Rightarrow \Gamma' \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma'$$
$$K \ni \alpha :: (L, U, T) \qquad \text{`}tag \in U \Rightarrow \text{`}tag \colon \tau_1 \in T\,.$$

Since $v/\text{`}tag(p_1) = \varsigma$, we know $v = \text{`}tag(v_1)$. Hence, by Lemma A.1, we have $\text{`}tag \in L$ and $K; \Gamma \vdash_\kappa v_1 \colon \tau_1'$ with $\text{`}tag \colon \tau_1' \in T$. Since $\text{`}tag \in U$, we also have $\text{`}tag \colon \tau_1 \in T$ and hence $\tau_1 = \tau_1'$ (as `tag is also in L and can only have a single type in T).

We therefore know $K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma'$ and $K; \Gamma \vdash_\kappa v_1 \colon \tau_1$, as well as $v_1/p_1 = \varsigma$. We can apply the induction hypothesis to conclude.

Cases TPk-And and TPk-Or Straightforward application of the induction hypothesis, to both sub-patterns for intersections and to the one that is actually selected for unions. □

LEMMA A.3: Stability of environment generation under type substitutions *If $K \vdash p \colon \tau \Rightarrow \Gamma$, then $K' \vdash p \colon \tau\theta \Rightarrow \Gamma\theta$ for every type substitution θ such that $K \vdash \theta \colon K'$.*

Proof By induction on the derivation of $K \vdash p \colon \tau \Rightarrow \Gamma$. We reason by cases on the last applied rule.

Cases TPk-Wildcard, TPk-Var, and TPk-Const Straightforward.

Case TPk-Pair We have

$$K \vdash (p_1, p_2) \colon \tau_1 \times \tau_2 \Rightarrow \Gamma_1 \cup \Gamma_2 \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma_1 \qquad K \vdash p_2 \colon \tau_2 \Rightarrow \Gamma_2\,.$$

By the induction hypothesis we derive both $K' \vdash p_1 \colon \tau_1\theta \Rightarrow \Gamma_1\theta$ and $K' \vdash p_2 \colon \tau_2\theta \Rightarrow \Gamma_2\theta$, then we apply *TPk-Pair* again to conclude.

Case TPk-Tag We have

$$K \vdash \text{`}tag(p_1) \colon \alpha \Rightarrow \Gamma \qquad K \vdash p_1 \colon \tau_1 \Rightarrow \Gamma \qquad K \ni \alpha :: (L, U, T) \qquad \text{`}tag \in U \Rightarrow \text{`}tag \colon \tau_1 \in T\,.$$

By the induction hypothesis we derive $K' \vdash p_1 \colon \tau_1\theta \Rightarrow \Gamma\theta$. Since $K \vdash \theta \colon K'$, $\alpha\theta$ must be a variable β such that $\beta :: (L', U', T') \in K'$. To apply *TPk-Tag* and conclude, we must establish that, if $\text{`}tag \in U'$, then $\text{`}tag \colon \tau_1\theta \in T'$. Since admissibility also implies $(L', U', T') \vDash (L, U, T\theta)$, we have $U' \subseteq U$ and $T\theta \subseteq T'$. Hence, if $\text{`}tag \in U'$, then $\text{`}tag \in U$, in which case $\text{`}tag \colon \tau_1 \in T$ and therefore $\text{`}tag \colon \tau_1\theta \in T\theta$ and hence $\text{`}tag \colon \tau_1\theta \in T'$.

Cases TPk-And and TPk-Or Straightforward application of the induction hypothesis, analogously to the case of pair patterns. □

LEMMA A.4: Stability of exhaustiveness under type substitutions *If $\tau \preccurlyeq_K P$, then $\tau\theta \preccurlyeq_{K'} P$ for any type substitution θ such that $K \vdash \theta \colon K'$.*

Proof We must prove, for every K'', θ' such that $K' \vdash \theta' \colon K''$ and every v such that $K''; \varnothing \vdash_\kappa v \colon \tau\theta\theta'$, that there exists a $p \in P$ which accepts v. This holds because $\theta' \circ \theta$ is such that $K \vdash \theta' \circ \theta \colon K''$: for any $\alpha :: (L, U, T) \in K$, we have $\alpha\theta :: (L', U', T') \in K'$ and hence $\alpha\theta\theta' :: (L'', U'', T'') \in K''$; we have $(L', U', T') \vDash (L, U, T\theta)$ and $(L'', U'', T'') \vDash (L', U', T'\theta')$ and therefore $(L'', U'', T'') \vDash (L, U, T\theta\theta')$. The conclusion follows by the definition of $\tau \preccurlyeq_K P$. □

Generalization

LEMMA A.5 *If* $\mathit{var}_K(\Gamma_1) \subseteq \mathit{var}_K(\Gamma_2)$*, then, for every type* τ, $\mathit{gen}_{K;\Gamma_1}(\tau) \sqsubseteq_K \mathit{gen}_{K;\Gamma_2}(\tau)$.

Proof An instance of $\mathit{gen}_{K;\Gamma_2}(\tau)$ is a type $\tau\theta$ such that $\mathit{dom}(\theta) \subseteq \mathit{var}_K(\tau) \setminus \mathit{var}_K(\Gamma_2)$ and $K \vdash \theta : K$. It is also an instance of $\mathit{gen}_{K;\Gamma_1}(\tau)$, with the same θ, since $\mathit{var}_K(\tau) \setminus \mathit{var}_K(\Gamma_2) \subseteq \mathit{var}_K(\tau) \setminus \mathit{var}_K(\Gamma_1)$. □

Properties of typing

LEMMA A.6: Weakening *Let K be a kinding environment and Γ_1, Γ_2 two type environments such that $\Gamma_1 \sqsubseteq_K \Gamma_2$ and $\mathit{var}_K(\Gamma_1) \subseteq \mathit{var}_K(\Gamma_2)$. If $K; \Gamma_2 \vdash_\kappa e : \tau$, then $K; \Gamma_1 \vdash_\kappa e : \tau$.*

Proof By induction on the derivation of $K; \Gamma_2 \vdash_\kappa e : \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have:

$$K; \Gamma_2 \vdash_\kappa x : \tau \qquad \tau \in \mathit{inst}_K(\Gamma_2(x))$$

and hence, since $\Gamma_1 \sqsubseteq_K \Gamma_2$, we have $\tau \in \mathit{inst}_K(\Gamma_1(x))$ and apply *Tk-Var* to conclude.

Case Tk-Const Straightforward.

Case Tk-Abstr We have:

$$K; \Gamma_2 \vdash_\kappa \lambda x.e_1 : \tau_1 \to \tau_2 \qquad K; \Gamma_2, \{x : \tau_1\} \vdash_\kappa e_1 : \tau_2 \,.$$

Since $\Gamma_1 \sqsubseteq_K \Gamma_2$, we have $\Gamma_1, \{x : \tau_1\} \sqsubseteq_K \Gamma_2, \{x : \tau_1\}$, and, since $\mathit{var}_K(\Gamma_1) \subseteq \mathit{var}_K(\Gamma_2)$, we have $\mathit{var}_K(\Gamma_1, \{x : \tau_1\}) \subseteq \mathit{var}_K(\Gamma_2, \{x : \tau_1\})$. Thus we may derive $K; \Gamma_1, \{x : \tau_1\} \vdash_\kappa e_1 : \tau_2$ by the induction hypothesis and apply *Tk-Abstr* to conclude.

Cases Tk-Appl, Tk-Pair, and Tk-Tag Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$K; \Gamma_2 \vdash_\kappa \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} : \tau$$
$$K; \Gamma_2 \vdash_\kappa e_0 : \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\ \ K \vdash p_i : \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma_2, \mathit{gen}_{K;\Gamma_2}(\Gamma_i) \vdash_\kappa e_i : \tau \ .$$

By the induction hypothesis, we derive $K; \Gamma_1 \vdash_\kappa e_0 : \tau_0$.

For every branch, note that by Lemma A.5 $\mathit{var}_K(\Gamma_1) \subseteq \mathit{var}_K(\Gamma_2)$ implies $\mathit{gen}_{K;\Gamma_1}(\tau) \sqsubseteq_K \mathit{gen}_{K;\Gamma_2}(\tau)$ for any τ. Hence, we have $\Gamma_1, \mathit{gen}_{K;\Gamma_1}(\Gamma_i) \sqsubseteq_K \Gamma_2, \mathit{gen}_{K;\Gamma_2}(\Gamma_i)$. Additionally, since $\mathit{var}_K(\mathit{gen}_{K;\Gamma_1}(\Gamma_i)) \subseteq \mathit{var}_K(\Gamma_1)$, we have $\mathit{var}_K(\Gamma_1, \mathit{gen}_{K;\Gamma_1}(\Gamma_i)) \subseteq \mathit{var}_K(\Gamma_2, \mathit{gen}_{K;\Gamma_2}(\Gamma_i))$.

Hence we may apply the induction hypothesis for all i to derive $K; \Gamma_1, \mathit{gen}_{K;\Gamma_1}(\Gamma_i) \vdash_\kappa e_i : \tau$ and then apply *Tk-Match* to conclude. □

LEMMA A.7: Stability of typing under type substitutions *Let K, K' be two closed, canonical kinding environments and θ a type substitution such that $K \vdash \theta : K'$. If $K; \Gamma \vdash_\kappa e : \tau$, then $K'; \Gamma\theta \vdash_\kappa e : \tau\theta$.*

Proof By induction on the derivation of $K; \Gamma \vdash_\kappa e : \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$K; \Gamma \vdash_{\kappa} x\colon \tau \qquad \tau \in \mathit{inst}_K(\Gamma(x))$$
$$\Gamma(x) = \forall A.\, K_x \triangleright \tau_x \qquad \tau = \tau_x\theta_x \qquad \mathit{dom}(\theta_x) \subseteq A \qquad K, K_x \vdash \theta_x\colon K$$

and must show

$$K'; \Gamma\theta \vdash_{\kappa} x\colon \tau\theta\,.$$

By α-renaming we can assume that θ does not involve A, that is, $A \cap \mathit{dom}(\theta) = \varnothing$ and $A \cap \mathit{var}_{\varnothing}(\theta) = \varnothing$, and also that $A \cap (\mathit{dom}(K') \cup \mathit{var}_{\varnothing}(K')) = \varnothing$, that is, that the variables in A are not assigned a kind in K' nor do they appear in the types in the typing component of the kinds in K'.

Under these assumptions, $(\Gamma\theta)(x) = \forall A.\, K_x\theta \triangleright \tau_x\theta$. We must show that $\tau\theta = \tau_x\theta\theta'_x$ for a substitution θ'_x such that $\mathit{dom}(\theta'_x) \subseteq A$ and $K', K_x\theta \vdash \theta'_x\colon K'$.

Let $\theta'_x = [\, {}^{\alpha\theta_x\theta}/_{\alpha} \mid \alpha \in A\,]$. First, we show that $\tau_x\theta\theta'_x = \tau_x\theta_x\theta = \tau\theta$, by showing that, for any α, $\alpha\theta\theta'_x = \alpha\theta_x\theta$. If $\alpha \in A$, then $\alpha\theta\theta'_x = \alpha\theta'_x = \alpha\theta_x\theta$ (θ is not defined on the variables in A). If $\alpha \notin A$, then $\alpha\theta\theta'_x = \alpha\theta$ (θ never produces any variable in A) and $\alpha\theta_x\theta = \alpha\theta$ as $\alpha \notin \mathit{dom}(\theta_x)$.

Since $\mathit{dom}(\theta'_x) \subseteq A$ holds, we only need to establish that $K', K_x\theta \vdash \theta'_x\colon K'$. This requires proving, for each $\alpha\colon\colon(L, U, T) \in K', K_x\theta$, that $\alpha\theta'_x$ is a type variable such that $\alpha\theta'_x\colon\colon(L', U', T') \in K'$ and $(L', U', T') \vDash (L, U, T\theta'_x)$.

Such an α can either be in the domain of $K_x\theta$ (if and only if it is in A) or in the domain of K'. In the latter case, we have $\alpha\theta'_x = \alpha$, since $\alpha \notin A$, and hence its kind in K' is the same as in $K', K_x\theta$. We must prove $(L, U, T) \vDash (L, U, T\theta'_x)$, which holds because the variables in A do not appear in T since $(L, U, T) \in K'$.

In the former case, we have $\alpha\colon\colon(L, U, T) \in K_x\theta$ and hence $\alpha\colon\colon(L, U, T_1) \in K_x$, with $T = T_1\theta$. Also, $\alpha\theta'_x = \alpha\theta_x\theta$. Since $K, K_x \vdash \theta_x\colon K$, $\alpha\theta_x\colon\colon(L_2, U_2, T_2) \in K$. Then, since $K \vdash \theta\colon K'$, $\alpha\theta_x\theta\colon\colon(L', U', T') \in K'$. We know $(L_2, U_2, T_2) \vDash (L, U, T_1\theta_x)$ and $(L', U', T') \vDash (L_2, U_2, T_2\theta)$. Both $L' \supseteq L$ and $U' \subseteq U$ hold by transitivity. We show $T' \supseteq T\theta'_x$ holds as well. If $\grave{}\mathit{tag}\colon \tau \in T\theta'_x$, since $T = T_1\theta$, then $\grave{}\mathit{tag}\colon \tau_1 \in T_1$ and $\tau = \tau_1\theta\theta'_x = \tau_1\theta_x\theta$. We thus have $\grave{}\mathit{tag}\colon \tau_1\theta_x \in T_1\theta_x$ and therefore $\grave{}\mathit{tag}\colon \tau_1\theta_x \in T_2$ and $\grave{}\mathit{tag}\colon \tau_1\theta_x\theta \in T'$.

Case Tk-Const Straightforward.

Case Tk-Abstr We have:

$$K; \Gamma \vdash_{\kappa} \lambda x.e_1\colon \tau_1 \to \tau_2 \qquad K; \Gamma, \{x\colon \tau_1\} \vdash_{\kappa} e_1\colon \tau_2\,.$$

By the induction hypothesis we have $K'; \Gamma\theta, \{x\colon \tau_1\theta\} \vdash_{\kappa} e_1\colon \tau_2\theta$. Then by *Tk-Abstr* we derive $K'; \Gamma\theta \vdash_{\kappa} \lambda x.e_1\colon (\tau_1 \to \tau_2)\theta$, since $(\tau_1 \to \tau_2)\theta = (\tau_1\theta) \to (\tau_2\theta)$.

Cases Tk-Appl and Tk-Pair Straightforward application of the induction hypothesis.

Case Tk-Match For the sake of clarity, we first prove the simpler case corresponding to (the encoding of) let, where – simplifying environment generation – we have

$$K; \Gamma \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ x \to e_1\colon \tau \qquad K; \Gamma \vdash_{\kappa} e_0\colon \tau_0 \qquad K; \Gamma, \mathit{gen}_{K;\Gamma}(\{x\colon \tau_0\}) \vdash_{\kappa} e_1\colon \tau$$

and must show

$$K'; \Gamma\theta \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ x \to e_1\colon \tau\theta$$

which we prove by establishing, for some type $\hat{\tau_0}$, that

$$K'; \Gamma\theta \vdash_{\kappa} e_0 \colon \hat{\tau_0} \qquad K'; \Gamma\theta, gen_{K';\Gamma\theta}(\{x \colon \hat{\tau_0}\}) \vdash_{\kappa} e_1 \colon \tau\theta \; .$$

Let $A = \{\alpha_1, \ldots, \alpha_n\} = var_K(\tau_0) \setminus var_K(\Gamma)$. We assume that the variables in A do not appear in the kinds of variables not in A, that is, that if $\alpha :: (L, U, T) \in K$ and $\alpha \notin A$, then $var_\varnothing(T) \cap A = \varnothing$.

This assumption is justified by the following observations. The variables in A only appear quantified in the environment used for the typing derivation for e_1. Therefore we may assume that they do not appear in τ: if they do, it is because they have been chosen when instantiating some type scheme and, since K is canonical, we might have chosen some other variable of the same kind. As for the occurrences of the variables in A in the derivation for e_0, a similar reasoning applies. These variables do not appear free in the environment (neither directly in a type in Γ, nor in the kinds of variables which appear free in Γ). Therefore, if they occur in τ_0 it is because they have been chosen either during instantiation of a type scheme or when typing an abstraction, and in both cases we might have chosen a different variable.

Now we rename these variables so that θ will not have effect on them. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \cap (dom(\theta) \cup var_\varnothing(\theta)) = \varnothing$ and $B \cap var_\varnothing(\Gamma) = \varnothing$. Let $\theta_0 = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ and $\theta' = \theta \circ \theta_0$. Since K' is canonical, we can choose each β_i so that, if $\alpha_i :: \bullet \in K$, then $\beta_i :: \bullet \in K'$, and if $\alpha_i :: (L, U, T) \in K$, then $\beta_i :: (L, U, T\theta') \in K$. As for A, we choose B so that the kinds in K' for variables not in B do not contain variables of B.

We show $K \vdash \theta' \colon K'$. For each $\alpha :: (L, U, T) \in K$, if $\alpha \in A$ then $\alpha = \alpha_i$ for some i, $\alpha\theta' = \beta_i$ and kind entailment holds straightforwardly by our choice of β_i. If $\alpha \notin A$, then $\alpha\theta' = \alpha\theta$ and the admissibility of θ implies $\alpha\theta :: (L', U', T') \in K'$ and $(L', U', T') \vDash (L, U, T\theta)$. We have $T\theta = T\theta'$ because of our assumption on A.

Since θ' is admissibile, by the induction hypothesis applied to θ', we derive $K; \Gamma\theta' \vdash_{\kappa} e_0 \colon \tau_0\theta'$. Since the variables in A do not appear in Γ, we have $\Gamma\theta' = \Gamma\theta$. We choose $\hat{\tau_0}$ to be $\tau_0\theta'$.

We apply the induction hypothesis to the derivation for e_1, this time using θ as the substitution. Now we have:

$$K'; \Gamma\theta \vdash_{\kappa} e_0 \colon \tau_0\theta' \qquad K'; \Gamma\theta, (gen_{K;\Gamma}(\{x \colon \tau_0\}))\theta \vdash_{\kappa} e_1 \colon \tau\theta \; .$$

We apply weakening (Lemma A.6) to derive from the latter the typing we need, that is,

$$K'; \Gamma\theta, gen_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\}) \vdash_{\kappa} e_1 \colon \tau\theta \; .$$

To do so we must show

$$\Gamma\theta, gen_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\}) \sqsubseteq_{K'} \Gamma\theta, (gen_{K;\Gamma}(\{x \colon \tau_0\})\}\theta$$
$$var_{K'}(\Gamma\theta, gen_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\})) \subseteq var_{K'}(\Gamma\theta, (gen_{K;\Gamma}(\{x \colon \tau_0\})\})\theta) \; .$$

The latter holds because $var_{K'}(\Gamma\theta, gen_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\})) \subseteq var_{K'}(\Gamma\theta)$.
As for the former, we prove $gen_{K';\Gamma\theta}(\{x \colon \tau_0\theta'\}) \sqsubseteq_{K'} (gen_{K;\Gamma}(\{x \colon \tau_0\})\})\theta$. We have

$$gen_{K;\Gamma}(\{x \colon \tau_0\}) \;=\; \forall A.\, K_x \triangleright \tau_0 \qquad K_x \;=\; \{\, \alpha :: \kappa \in K \mid \alpha \in A \,\} \; .$$

By α-renaming of the quantified variables we can write

$$gen_{K;\Gamma}(\{x \colon \tau_0\}) \;=\; \forall B.\, K_x^* \triangleright \tau_0\theta_0$$
$$K_x^* \;=\; \{\, \beta_i :: \bullet \mid \alpha_i :: \bullet \in K_x \,\} \cup \{\, \beta_i :: (L, U, T\theta_0) \mid \alpha_i :: (L, U, T) \in A \,\}$$

and, since θ does not involve B,

$$(gen_{K;\Gamma}(\{x\colon \tau_0\}))\theta \;=\; \forall B.\, K_x^*\theta \triangleright \tau_0\theta_0\theta \;=\; \forall B.\, K'_x \triangleright \tau_0\theta'$$
$$K'_x \;=\; \{\, \beta :: \kappa \in K' \mid \beta \in B \,\}\,.$$

The other type scheme is

$$gen_{K';\Gamma\theta}(\tau_0\theta') \;=\; \forall C.\, K'_C \triangleright \tau_0\theta'$$
$$C \;=\; var_{K'}(\tau_0\theta') \setminus var_{K'}(\Gamma\theta) \qquad K'_C \;=\; \{\, \beta :: \kappa \in K' \mid \beta \in C \,\}\,.$$

We show $B \subseteq C$, which concludes the proof (because the kinding environments are both restrictions of K'). Consider $\beta_i \in B$. We have $\alpha_i \in var_K(\tau_0) \setminus var_K(\Gamma)$. Then $\beta_i = \alpha_i\theta' \in var_{K'}(\tau_0\theta')$. Furthermore $\beta_i \notin var_{K'}(\Gamma\theta)$ holds because $\Gamma\theta$ does not contain variables in B (Γ does not contain them and θ does not introduce them) and variables in B do not appear in the kinds of other variables which are not themselves in B.

We now consider the rule *Tk-Match* in its generality. We have

$$K;\Gamma \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \tau$$
$$K;\Gamma \vdash_{\kappa} e_0\colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\;\; K \vdash p_i\colon \tau_0 \Rightarrow \Gamma_i \qquad K;\Gamma, gen_{K;\Gamma}(\Gamma_i) \vdash_{\kappa} e_i\colon \tau$$

and must show

$$K';\Gamma\theta \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \tau\theta$$

which we prove by establishing, for some $\hat{\tau_0}$ and $\{\, \hat{\Gamma_i} \mid i \in I \,\}$, that

$$K';\Gamma\theta \vdash_{\kappa} e_0\colon \hat{\tau_0} \qquad \hat{\tau_0} \preccurlyeq_{K'} \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\;\; K' \vdash p_i\colon \hat{\tau_0} \Rightarrow \hat{\Gamma_i} \qquad K';\Gamma\theta, gen_{K';\Gamma\theta}(\hat{\Gamma_i}) \vdash_{\kappa} e_i\colon \tau\theta\;\;.$$

For the derivation for e_0 we proceed as above and have $\hat{\tau_0} = \tau_0\theta'$. By Lemma A.4 we have $\tau_0\theta' \preccurlyeq_{K'} \{\, p_i \mid i \in I \,\}$. By Lemma A.3, we have $K' \vdash p_i\colon \tau_0\theta' \Rightarrow \Gamma_i\theta'$ and thus take $\hat{\Gamma_i} = \Gamma_i\theta'$.

We proceed as before also for the derivations for each branch. The difference is that, to apply weakening, we must prove the two premises for the environments and not for τ_0 alone. The condition on variables is straightforward, as before. For the other we prove, for each $x \in capt(p_i)$ and assuming $\Gamma_i(x) = \tau_x$,

$$\Gamma\theta, gen_{K';\Gamma\theta}(\tau_x\theta') \sqsubseteq_{K'} \Gamma\theta, (gen_{K;\Gamma}(\tau_x))\theta\,.$$

We show it as for τ_0 above: $var_K(\tau_x)$ is always a subset of $var_K(\tau_0)$ because environment generation does not introduce new variables. □

LEMMA A.8: Expression substitution *Let $x_1, \ldots, x_n$ be distinct variables and $v_1, \ldots, v_n$ values. Let $\Gamma' = \{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}$ and $\varsigma = [v_1/x_1, \ldots, v_n/x_n]$.*

If $K;\Gamma,\Gamma' \vdash_{\kappa} e\colon \tau$ and, for all $k \in \{1, \ldots, n\}$ and for all $\tau_k \in inst_K(\sigma_k)$, $K;\Gamma \vdash_{\kappa} v_k\colon \tau_k$, then $K;\Gamma \vdash_{\kappa} e\varsigma\colon \tau$.

Proof By induction on the derivation of $K;\Gamma,\Gamma' \vdash_{\kappa} e\colon \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$K; \Gamma, \Gamma' \vdash_{\kappa} x\colon \tau \qquad \tau \in \mathit{inst}_K((\Gamma, \Gamma')(x))\,.$$

Either $x = x_k$ for some k or not. In the latter case, $x\varsigma = x$, $x \notin \mathit{dom}(\Gamma')$ and hence $(\Gamma, \Gamma')(x) = \Gamma(x)$. Then, since $\tau \in \mathit{inst}_K((\Gamma, \Gamma')(x))$, $\tau \in \mathit{inst}_K(\Gamma(x))$ and *Tk-Var* can be applied.

If $x = x_k$, then $(\Gamma, \Gamma')(x) = \Gamma'(x) = \sigma_k$. We must then prove $K; \Gamma \vdash_{\kappa} v_k\colon \tau$, which we know by hypothesis since $\tau \in \mathit{inst}_K(\sigma_k)$.

Case Tk-Const Straightforward.

Case Tk-Abstr We have

$$K; \Gamma, \Gamma' \vdash_{\kappa} \lambda x.e_1\colon \tau_1 \to \tau_2 \qquad K; \Gamma, \Gamma', \{x\colon \tau_1\} \vdash_{\kappa} e_1\colon \tau_2\,.$$

By α-renaming we can assume $x \notin \mathit{dom}(\Gamma')$; then $(\lambda x.e_1)\varsigma = \lambda x.(e_1\varsigma)$ and $\Gamma, \Gamma', \{x\colon \tau_1\} = \Gamma, \{x\colon \tau_1\}, \Gamma'$. Therefore we have $K; \Gamma, \{x\colon \tau_1\}, \Gamma' \vdash_{\kappa} e_1\colon \tau_2$ and, by the induction hypothesis, $K; \Gamma, \{x\colon \tau_1\} \vdash_{\kappa} e_1\colon \tau_2$. We apply *Tk-Abstr* to conclude.

Cases Tk-Appl, Tk-Pair, and Tk-Tag Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$\begin{aligned}
&K; \Gamma, \Gamma' \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I}\colon \tau\\
&K; \Gamma, \Gamma' \vdash_{\kappa} e_0\colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}\\
&\forall\, i \in I.\ \ K \vdash p_i\colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \Gamma', \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) \vdash_{\kappa} e_i\colon \tau\ \ .
\end{aligned}$$

We assume by α-renaming that no capture variable of any pattern is in the domain of Γ'. Then, $(\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I})\varsigma = \mathsf{match}\ e_0\varsigma\ \mathsf{with}\ (p_i \to e_i\varsigma)_{i \in I}$ and $\Gamma, \Gamma', \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) = \Gamma, \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i), \Gamma'$ for any i.

By the induction hypothesis, we derive $K; \Gamma \vdash_{\kappa} e_0\varsigma\colon \tau_0$ and $K; \Gamma, \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i) \vdash_{\kappa} e_i\colon \tau$ for all i. From the latter, we prove $K; \Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\kappa} e_i\colon \tau$ by weakening (Lemma A.6): we have $\mathit{gen}_{K;\Gamma}(\Gamma_i) \sqsubseteq_K \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i)$ by Lemma A.5 – since $\mathit{var}_K(\Gamma) \subseteq \mathit{var}_K(\Gamma, \Gamma')$ – and clearly we have $\mathit{var}_K(\Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_i)) \subseteq \mathit{var}_K(\Gamma, \mathit{gen}_{K;\Gamma,\Gamma'}(\Gamma_i))$ since $\mathit{var}_K(\mathit{gen}_{K;\Gamma}(\Gamma_i)) \subseteq \mathit{var}_K(\Gamma)$. □

Type soundness

THEOREM A.9: Progress *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof By hypothesis we have $K; \varnothing \vdash_{\kappa} e\colon \tau$. The proof is by induction on its derivation; we reason by cases on the last applied rule.

Case Tk-Var This case does not occur because variables are not closed.

Case Tk-Const In this case e is a constant c and therefore a value.

Case Tk-Abstr In this case e is an abstraction $\lambda x.e_1$. Since it is also closed, it is a value.

Case Tk-Appl We have

$$K; \varnothing \vdash_{\kappa} e_1\ e_2 \colon \tau \qquad K; \varnothing \vdash_{\kappa} e_1 \colon \tau' \to \tau \qquad K; \varnothing \vdash_{\kappa} e_2 \colon \tau' \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $e_1\ e_2 \rightsquigarrow e_1'\ e_2$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $e_1\ e_2 \rightsquigarrow e_1\ e_2'$.

If both are values then, by Lemma A.1, e_1 has the form $\lambda x.e_3$ for some e_3. Then, we can apply *R-Appl* and $e_1\ e_2 \rightsquigarrow e_3[e_2/x]$.

Case Tk-Pair We have

$$K; \varnothing \vdash_{\kappa} (e_1, e_2) \colon \tau_1 \times \tau_2 \qquad K; \varnothing \vdash_{\kappa} e_1 \colon \tau_1 \qquad K; \varnothing \vdash_{\kappa} e_2 \colon \tau_2 \,.$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e_1'$, then $(e_1, e_2) \rightsquigarrow (e_1', e_2)$. If e_1 is a value and $e_2 \rightsquigarrow e_2'$, then $(e_1, e_2) \rightsquigarrow (e_1, e_2')$. If both are values, then (e_1, e_2) is also a value.

Case Tk-Tag We have

$$K; \varnothing \vdash_{\kappa} \text{`}tag(e_1) \colon \alpha \qquad K; \varnothing \vdash_{\kappa} e_1 \colon \tau_1 \,.$$

Analogously to the previous case, by the induction hypothesis we have that either e_1 is a value or $e_1 \rightsquigarrow e_1'$. In the former case, $\text{`}tag(e_1)$ is a value as well. In the latter, we have $\text{`}tag(e_1) \rightsquigarrow \text{`}tag(e_1')$.

Case Tk-Match We have

$$K; \varnothing \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \colon \tau \qquad K; \varnothing \vdash_{\kappa} e_0 \colon \tau_0 \qquad \tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\} \,.$$

By the inductive hypothesis, either e_0 is a value or it may reduce. In the latter case, if $e_0 \rightsquigarrow e_0'$, then $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow \mathsf{match}\ e_0'\ \mathsf{with}\ (p_i \to e_i)_{i \in I}$.

If e_0 is a value, on the other hand, the expression may reduce by application of *R-Match*. Since $\tau_0 \preccurlyeq_K \{\, p_i \mid i \in I \,\}$ and e_0 is a value of type τ_0 (and therefore satisfies the premises of the definition of exhaustiveness, with $\theta = [\,]$ and $K = K'$), there exists at least an $i \in I$ such that $e_0 / p_i = \varsigma$ for some substitution ς. Let j be the least of these i and ς_j the corresponding substitution; then $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow e_j \varsigma_j$. □

THEOREM A.10: Subject reduction *Let e be an expression and τ a type such that $K; \Gamma \vdash_{\kappa} e \colon \tau$. If $e \rightsquigarrow e'$, then $K; \Gamma \vdash_{\kappa} e' \colon \tau$.*

Proof By induction on the derivation of $K; \Gamma \vdash_{\kappa} e \colon \tau$. We reason by cases on the last applied rule.

Cases Tk-Var, Tk-Const, and Tk-Abstr These cases may not occur: variables, constants, and abstractions never reduce.

Case Tk-Appl We have

$$K; \Gamma \vdash_{\kappa} e_1\ e_2 \colon \tau \qquad K; \Gamma \vdash_{\kappa} e_1 \colon \tau' \to \tau \qquad K; \Gamma \vdash_{\kappa} e_2 \colon \tau' \,.$$

$e_1\ e_2 \rightsquigarrow e'$ occurs in any of three ways: *i*) $e_1 \rightsquigarrow e_1'$ and $e' = e_1'\ e_2$; *ii*) e_1 is a value, $e_2 \rightsquigarrow e_2'$ and $e' = e_1\ e_2'$; *iii*) both e_1 and e_2 are values, e_1 is of the form $\lambda x.e_3$, and $e' = e_3[e_2/x]$.

In the first case, we derive by the induction hypothesis that $K; \Gamma \vdash_{\kappa} e_1' \colon \tau' \to \tau$ and conclude by applying *Tk-Appl* again. The second case is analogous.

In the third case, we know by Lemma A.1 that $K; \Gamma, \{x \colon \tau'\} \vdash_{\kappa} e_3 \colon \tau$. We also know that e_2 is a value such that $K; \Gamma \vdash_{\kappa} e_2 \colon \tau'$. Then, by Lemma A.8, $K; \Gamma \vdash_{\kappa} e_3[e_2/x] \colon \tau$.

Case Tk-Pair We have

$$K; \Gamma \vdash_{\kappa} (e_1, e_2) \colon \tau_1 \times \tau_2 \qquad K; \Gamma \vdash_{\kappa} e_1 \colon \tau_1 \qquad K; \Gamma \vdash_{\kappa} e_2 \colon \tau_2 \,.$$

$(e_1, e_2) \rightsquigarrow e'$ occurs either because $e_1 \rightsquigarrow e_1'$ and $e' = (e_1', e_2)$, or because e_1 is a value, $e_2 \rightsquigarrow e_2'$, and $e' = (e_1, e_2')$. In either case, the induction hypothesis allows us to derive that the type of the component that reduces is preserved; therefore, we can apply *Tk-Pair* again to conclude.

Case Tk-Tag Analogously to the previous case, a variant expression only reduces if its argument does, so we apply the induction hypothesis and *Tk-Tag* to conclude.

Case Tk-Match We have

$$K; \Gamma \vdash_{\kappa} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \colon \tau$$
$$K; \Gamma \vdash_{\kappa} e_0 \colon \tau_0 \qquad \forall\, i \in I.\ K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_i) \vdash_{\kappa} e_i \colon \tau \ .$$

$\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \rightsquigarrow e'$ occurs either because $e_0 \rightsquigarrow e_0'$ and $e' = \mathsf{match}\ e_0'\ \mathsf{with}\ (p_i \to e_i)_{i \in I}$ or because e_0 is a value and $e' = e_j \varsigma$, where $e_0 / p_j = \varsigma$ and, for all $i < j$, $e_0 / p_i = \Omega$. In the former case, we apply the induction hypothesis and conclude by *Tk-Match*.

In the latter case, ς is a substitution from the capture variables of p_j to values, and we know by Lemma A.2 that, for all $x \in \mathit{capt}(p_j)$, $K; \Gamma \vdash_{\kappa} x\varsigma \colon \Gamma_j(x)$. We show that, additionally, $K; \Gamma \vdash_{\kappa} x\varsigma \colon \tau_x$ holds for every $\tau_x \in \mathit{inst}_K(\mathit{gen}_{K;\Gamma}(\Gamma_j(x)))$. Every such τ_x is equal to $\Gamma_j(x)\theta$ for a θ such that $\mathit{dom}(\theta) \subseteq \mathit{var}_K(\Gamma_j(x)) \setminus \mathit{var}_K(\Gamma)$ and $K \vdash \theta \colon K$ (the kinding environment captured by generalization is just a subset of K). Then, $K; \Gamma \vdash_{\kappa} x\varsigma \colon \Gamma_j(x)\theta$ holds by Lemma A.7, since $\Gamma\theta = \Gamma$ (the substitution does not change any free variable of Γ).

From $K; \Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_j) \vdash_{\kappa} e_j \colon \tau$ and from the fact that we have $K; \Gamma \vdash_{\kappa} x\varsigma \colon \tau_x$ for all $x \in \mathit{dom}(\Gamma_j)$ and all $\tau_x \in \mathit{inst}_K(\mathit{gen}_{K;\Gamma}(\Gamma_j(x)))$, we derive $K; \Gamma \vdash_{\kappa} e_j\varsigma \colon \tau$ by Lemma A.8. □

COROLLARY A.11: Type soundness *Let e be a well-typed, closed expression, that is, such that $K; \varnothing \vdash_{\kappa} e \colon \tau$ holds for some τ. Then, either e diverges or it reduces to a value v such that $K; \varnothing \vdash_{\kappa} v \colon \tau$.*

Proof Consequence of Theorem A.9 and Theorem A.10. □

A.2 VARIANTS WITH SET-THEORETIC TYPES

LEMMA A.12: Generation for values *Let v be a value. Then:*

- *if $\Gamma \vdash_{\mathrm{s}} v \colon c$, then $v = c$;*
- *if $\Gamma \vdash_{\mathrm{s}} v \colon b$, then $v = c$ for some constant c such that $b_c \leq b$;*
- *if $\Gamma \vdash_{\mathrm{s}} v \colon t_1 \to t_2$, then v is of the form $\lambda x.e$ and $\Gamma, \{x \colon t_1\} \vdash_{\mathrm{s}} e \colon t_2$;*
- *if $\Gamma \vdash_{\mathrm{s}} v \colon t_1 \times t_2$, then v is of the form (v_1, v_2), $\Gamma \vdash_{\mathrm{s}} v_1 \colon t_1$, and $\Gamma \vdash_{\mathrm{s}} v_2 \colon t_2$;*
- *if $\Gamma \vdash_{\mathrm{s}} v \colon `tag(t_1)$, then $v$ is of the form $`tag(v_1)$ and $\Gamma \vdash_{\mathrm{s}} v_1 \colon t_1$.*

Proof By induction on the typing derivation: values must be typed by an application of the rule corresponding to their form, to appropriate premises, possibly followed by applications of *Ts-Subsum*.

The base cases are straightforward. In the inductive step, we just apply the induction hypothesis; for abstractions, the result follows from the behaviour of subtyping on arrow types. □

Patterns and environment generation

We state the three lemmas below without proof, as they rely on the model of types which we have not discussed. Details can be found in Frisch, Castagna, and Benzaken (2008), Frisch (2004), and Castagna and Xu (2011).

LEMMA A.13 *For each $i \in I$, let p_i be a pattern. If $\Gamma \vdash_s v\colon \bigvee_{i\in I} \lfloor p_i \rfloor$, then there exists an $i \in I$ such that $\Gamma \vdash_s v\colon \lfloor p_i \rfloor$.*

LEMMA A.14 *Let t be a type. Let t' be a type such that either $t' = \lfloor p \rfloor$ or $t' = \neg \lfloor p \rfloor$, for some pattern p. If $\Gamma \vdash_s v\colon t$ and $\Gamma \vdash_s v\colon t'$, then $\Gamma \vdash_s v\colon t \wedge t'$.*

LEMMA A.15 *Let v be a well-typed value* (i.e. $\varnothing \vdash_s v\colon t$ *holds for some t) and p a pattern. Then:*

- *$\varnothing \vdash_s v\colon \lfloor p \rfloor$ holds if and only if $v/p = \varsigma$ for some substitution ς;*
- *$\varnothing \vdash_s v\colon \neg\lfloor p \rfloor$ holds if and only if $v/p = \Omega$.*

LEMMA A.16 *Let p be a pattern and t, t' two types. If $t \leq t' \leq \lfloor p \rfloor$, then, for all $x \in \mathit{capt}(p)$, $(t/\!/p)(x) \leq (t'/\!/p)(x)$.*

Proof By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathit{capt}(p) = \varnothing$.

Case $p = x$ We must prove $(t/\!/x)(x) \leq (t'/\!/x)(x)$, that is, $t \leq t'$, which we know by hypothesis.

Case $p = (p_1, p_2)$ Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, $(t/\!/p)(x) = (\pi_i(t)/\!/p_i)(x)$ and $(t'/\!/p)(x) = (\pi_i(t')/\!/p_i)(x)$. Since $t \leq t'$ implies $\pi_i(t) \leq \pi_i(t')$ by Property 4.10, we can apply the induction hypothesis to conclude.

Case $p = \backslash tag(p)$ Analogous to the previous case, because $t \leq t'$ implies $\pi_{\backslash tag}(t) \leq \pi_{\backslash tag}(t')$ by Property 4.11.

Case $p = p_1 \& p_2$ Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, $(t/\!/p)(x) = (t/\!/p_i)(x)$ and $(t'/\!/p)(x) = (t'/\!/p_i)(x)$. We apply the induction hypothesis to conclude.

Case $p = p_1 | p_2$ Every $x \in \mathit{capt}(p)$ is both in $\mathit{capt}(p_1)$ and in $\mathit{capt}(p_2)$. We have that $(t/\!/p)(x) = (t \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor /\!/ p_2)(x)$ and likewise for t'. Since $t \wedge \lfloor p_1 \rfloor \leq t' \wedge \lfloor p_1 \rfloor$ and $t \setminus \lfloor p_1 \rfloor \leq t' \setminus \lfloor p_1 \rfloor$, we can apply the induction hypothesis to both sub-patterns to derive $(t \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \leq (t' \wedge \lfloor p_1 \rfloor /\!/ p_1)(x)$ and $(t \setminus \lfloor p_1 \rfloor /\!/ p_2)(x) \leq (t' \setminus \lfloor p_1 \rfloor /\!/ p_2)(x)$. Then we have $(t \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor /\!/ p_2)(x) \leq (t' \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \vee (t' \setminus \lfloor p_1 \rfloor /\!/ p_2)(x)$ □

LEMMA A.17: Correctness of environment generation *Let p be a pattern and v a value such that $\Gamma \vdash_s v\colon t$ for some $t \leq \lfloor p \rfloor$. Then, for all $x \in \mathit{capt}(p)$, $\Gamma \vdash_s x(v/p)\colon (t/\!/p)(x)$.*

Proof By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathit{capt}(p) = \varnothing$.

Case $p = x$ We must prove $\Gamma \vdash_s x[v/x]\colon (t/\!/x)(x)$, which is the hypothesis $\Gamma \vdash_s v\colon t$.

Case $p = (p_1, p_2)$ We have $t \leq \mathbb{1} \times \mathbb{1}$, hence $t \leq \pi_1(t) \times \pi_2(t)$; then, since $\Gamma \vdash_s v\colon \pi_1(t) \times \pi_2(t)$ by subsumption, we have by Lemma A.12 that $v = (v_1, v_2)$ and that $\Gamma \vdash_s v_i\colon \pi_1(t)$ for both i. Furthermore, $t \leq \lfloor (p_1, p_2) \rfloor = \lfloor p_1 \rfloor \times \lfloor p_2 \rfloor$. Hence, by Property 4.10, $\pi_i(t) \leq \lfloor p_i \rfloor$ for both i.

Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, $x(v/p) = x(v_i/p_i)$ and $(t/\!/p)(x) = (\pi_i(t)/\!/p_i)(x)$. We apply the induction hypothesis to conclude.

Case $p = {}^{\backprime}\mathit{tag}(p)$ Analogous to the previous case.

Case $p = p_1 \& p_2$ Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, we can directly apply the induction hypothesis since $t \leq \lfloor p_1 \& p_2 \rfloor$ implies $t \leq \lfloor p_1 \rfloor$ and $t \leq \lfloor p_2 \rfloor$.

Case $p = p_1 | p_2$ Either $v/p = v/p_1$ or $v/p = v/p_2$ (in which case $v/p_1 = \Omega$).

Case $v/p = v/p_1$ By Lemma A.15 we have $\Gamma \vdash_s v\colon \lfloor p_1 \rfloor$; by Lemma A.14 we have $\Gamma \vdash_s v\colon t \wedge \lfloor p_1 \rfloor$. Since $t \wedge \lfloor p_1 \rfloor \leq \lfloor p_1 \rfloor$, by the induction hypothesis we have, for all $x \in \mathit{capt}(p_1) = \mathit{capt}(p)$, $\Gamma \vdash_s x(v/p)\colon (t \wedge \lfloor p_1 \rfloor/\!/p_1)(x)$ and, by subsumption, $\Gamma \vdash_s x(v/p)\colon (t \wedge \lfloor p_1 \rfloor/\!/p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor/\!/p_2)(x)$.

Case $v/p = v/p_2$ By Lemma A.15 and Lemma A.14, we have $\Gamma \vdash_s v\colon t \setminus \lfloor p_1 \rfloor$. Additionally, $t \setminus \lfloor p_1 \rfloor \leq \lfloor p_2 \rfloor$ holds because it is equivalent to $t \leq \lfloor p_1 \rfloor \vee \lfloor p_2 \rfloor$. Therefore by the induction hypothesis we have, for all $x \in \mathit{capt}(p_1) = \mathit{capt}(p)$, $\Gamma \vdash_s x(v/p)\colon (t \setminus \lfloor p_1 \rfloor/\!/p_2)(x)$ and, by subsumption, $\Gamma \vdash_s x(v/p)\colon (t \wedge \lfloor p_1 \rfloor/\!/p_1)(x) \vee (t \setminus \lfloor p_1 \rfloor/\!/p_2)(x)$. □

LEMMA A.18 *Let p be a pattern, t a type such that $t \leq \lfloor p \rfloor$, and θ a type substitution. Then, for all $x \in \mathit{capt}(p)$, $(t\theta/\!/p)(x) \leq ((t/\!/p)(x))\theta$.*

Proof By structural induction on p.

Cases $p = \_$ *and* $p = c$ There is nothing to prove since $\mathit{capt}(p) = \varnothing$.

Case $p = x$ We must prove $(t\theta/\!/x)(x) \leq (t/\!/x)(x)\theta$, which is $t\theta \leq t\theta$.

Case $p = (p_1, p_2)$ Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, $(t\theta/\!/p)(x) = (\pi_i(t\theta)/\!/p_i)(x)$ and $(t/\!/p)(x)\theta = (\pi_i(t)/\!/p_i)(x)\theta$.

Since $\pi_i(t\theta) \leq \pi_i(t)\theta$, by Lemma A.16 we have $(\pi_i(t\theta)/\!/p_i)(x) \leq (\pi_i(t)\theta/\!/p_i)(x)$. By the induction hypothesis we have $(\pi_i(t)\theta/\!/p_i)(x) \leq (\pi_i(t)/\!/p_i)(x)\theta$.

Case $p = {}^{\backprime}\mathit{tag}(p)$ Analogous to the previous case, since $\pi_{{}^{\backprime}\mathit{tag}}(t\theta) \leq \pi_{{}^{\backprime}\mathit{tag}}(t)\theta$.

Case $p = p_1 \& p_2$ Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Assume $x \in \mathit{capt}(p_i)$; then, $(t\theta/\!/p)(x) = (t\theta/\!/p_i)(x)$ and $(t/\!/p)(x)\theta = (t/\!/p_i)(x)\theta$. We conclude by the induction hypothesis.

Case $p = p_1 | p_2$ Every $x \in \mathit{capt}(p)$ is both in $\mathit{capt}(p_1)$ and in $\mathit{capt}(p_2)$. We have $(t\theta/\!/p)(x) = ((t \wedge \lfloor p_1 \rfloor)\theta/\!/p_1)(x) \vee ((t \setminus \lfloor p_1 \rfloor)\theta/\!/p_2)(x)$ – pattern types are closed, so we can apply θ to them too – and $(t/\!/p)(x)\theta = (t \wedge \lfloor p_1 \rfloor/\!/p_1)(x)\theta \vee (t \setminus \lfloor p_1 \rfloor/\!/p_2)(x)\theta$. We conclude by applying the induction hypothesis to both members of the union. □

Generalization

LEMMA A.19 *Let Γ_1, Γ_2 be two type environments such that $\mathit{var}(\Gamma_1) \subseteq \mathit{var}(\Gamma_2)$ and t_1, t_2 two types such that $t_1 \leq t_2$. Then $\mathit{gen}_{\Gamma_1}(t_1) \sqsubseteq \mathit{gen}_{\Gamma_2}(t_2)$.*

Proof An instance of $\mathit{gen}_{\Gamma_2}(t_2)$ is a type $t_2\theta_2$ such that $\mathit{dom}(\theta_2) \subseteq \mathit{var}(t_2) \setminus \mathit{var}(\Gamma_2)$. Let θ_1 be the restriction of θ_2 to the variables in $\mathit{var}(t_1) \setminus \mathit{var}(\Gamma_1)$. Then, $t_1\theta_1$ is an instance of $\mathit{gen}_{\Gamma_1}(t_1)$.

We have $t_1\theta_1 = t_1\theta_2$ because the two substitutions differ only on variables in $\mathit{var}(t_2) \setminus \mathit{var}(t_1)$ (which do not appear in t_1 at all) or in $\mathit{var}(\Gamma_1) \setminus \mathit{var}(\Gamma_2)$ (which is empty). Finally, we have $t_1\theta_2 \leq t_2\theta_2$ because subtyping is preserved by substitutions. □

Properties of typing

LEMMA A.20: Weakening *Let Γ_1, Γ_2 be two type environments such that $\Gamma_1 \sqsubseteq \Gamma_2$ and $\mathit{var}(\Gamma_1) \subseteq \mathit{var}(\Gamma_2)$. If $\Gamma_2 \vdash_s e\colon t$, then $\Gamma_1 \vdash_s e\colon t$.*

Proof By induction on the derivation of $\Gamma_2 \vdash_s e\colon t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma_2 \vdash_s x\colon t \qquad t \in \mathit{inst}(\Gamma_2(x))$$

and hence, since $\Gamma_1 \sqsubseteq \Gamma_2$, there exists a $t' \in \mathit{inst}(\Gamma_1(x))$ such that $t' \leq t$. We apply *Ts-Var* to derive $\Gamma_1 \vdash_s x\colon t'$ and *Ts-Subsum* to conclude.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma_2 \vdash_s \lambda x.e_1\colon t_1 \to t_2 \qquad \Gamma_2, \{x\colon t_1\} \vdash_s e_1\colon t_2\,.$$

Since $\Gamma_1 \sqsubseteq \Gamma_2$, we have $\Gamma_1, \{x\colon t_1\} \sqsubseteq \Gamma_2, \{x\colon t_1\}$; since $\mathit{var}(\Gamma_1) \subseteq \mathit{var}(\Gamma_2)$, we have $\mathit{var}(\Gamma_1, \{x\colon t_1\}) \subseteq \mathit{var}(\Gamma_2, \{x\colon t_1\})$. We derive $\Gamma_1, \{x\colon t_1\} \vdash_s e_1\colon t_2$ by the induction hypothesis and apply *Ts-Abstr* to conclude.

Cases Ts-Appl, Ts-Pair, Ts-Tag, and Ts-Subsum Straightforward application of the induction hypothesis.

Case Tk-Match We have

$$\begin{array}{l}\Gamma_2 \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t\\ \Gamma_2 \vdash_s e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor\\ \forall i \in I \text{ s.t. } t_i \not\leq \mathbb{0}.\ \ \Gamma_2, \mathit{gen}_{\Gamma_2}(t_i /\!/ p_i) \vdash_s e_i\colon t_i' \qquad t = \bigvee_{t_i \not\leq \mathbb{0}} t_i'\ .\end{array}$$

By the induction hypothesis, we derive $\Gamma_1 \vdash_s e_0\colon t_0$.

For any branch, note that $\mathit{var}(\Gamma_1) \subseteq \mathit{var}(\Gamma_2)$ implies $\mathit{gen}_{\Gamma_1}(t) \sqsubseteq \mathit{gen}_{\Gamma_2}(t)$ for any t by Lemma A.19. Hence, we have $\Gamma_1, \mathit{gen}_{\Gamma_1}(t_i /\!/ p_i) \sqsubseteq \Gamma_2, \mathit{gen}_{\Gamma_2}(t_i /\!/ p_i)$. Additionally, since $\mathit{var}(\mathit{gen}_{\Gamma_1}(t_i /\!/ p_i)) \subseteq \mathit{var}(\Gamma_1) \subseteq \mathit{var}(\Gamma_2)$, we have $\mathit{var}(\Gamma_1, \mathit{gen}_{\Gamma_1}(t_i /\!/ p_i)) \subseteq \mathit{var}(\Gamma_2, \mathit{gen}_{\Gamma_2}(t_i /\!/ p_i))$.

Hence we may apply the induction hypothesis for all i to derive $\Gamma_1, \mathit{gen}_{\Gamma_1}(t_i /\!/ p_i) \vdash_s e_i\colon t_i'$ and then apply *Ts-Match* to conclude. □

LEMMA A.21: Stability of typing under type substitutions *Let θ be a type substitution. If $\Gamma \vdash_s e\colon t$, then $\Gamma\theta \vdash_s e\colon t\theta$.*

Proof By induction on the derivation of $\Gamma \vdash_s e\colon t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma \vdash_s x\colon t \qquad t \in \mathit{inst}(\Gamma(x)) \qquad \Gamma(x) = \forall A.\, t_x \qquad t = t_x\theta_x \qquad \mathit{dom}(\theta_x) \subseteq A$$

and must show $\Gamma\theta \vdash_s x\colon t\theta$.

By α-renaming we can assume $A \sharp \theta$, that is, $A \cap \mathit{dom}(\theta) = \varnothing$ and $A \cap \mathit{var}(\theta) = \varnothing$. Under this assumption, $(\Gamma\theta)(x) = \forall A.\, t_x\theta$. We must show that $t\theta = t_x\theta\theta'_x$ for a substitution θ'_x such that $\mathit{dom}(\theta'_x) \subseteq A$.

Let $\theta'_x = [\,{}^{\alpha\theta_x\theta}/_{\alpha} \mid \alpha \in A\,]$. We show that $t\theta\theta'_x = t_x\theta_x\theta = t\theta$, by showing that, for every α, $\alpha\theta\theta'_x = \alpha\theta_x\theta$. If $\alpha \in A$, then $\alpha\theta\theta'_x = \alpha\theta'_x = \alpha\theta_x\theta$ (θ is not defined on the variables in A). If $\alpha \notin A$, then $\alpha\theta\theta'_x = \alpha\theta$ (θ never produces any variable in A) and $\alpha\theta_x\theta = \alpha\theta$ as $\alpha \notin \mathit{dom}(\theta_x)$.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma \vdash_s \lambda x.e_1\colon t_1 \to t_2 \qquad \Gamma, \{x\colon t_1\} \vdash_s e_1\colon t_2\,.$$

By the induction hypothesis we have $\Gamma\theta, \{x\colon t_1\theta\} \vdash_s e_1\colon t_2\theta$. Then by *Ts-Abstr* we derive $\Gamma\theta \vdash_s \lambda x.e_1\colon (t_1\theta) \to (t_2\theta)$, which is $\Gamma\theta \vdash_s \lambda x.e_1\colon (t_1 \to t_2)\theta$.

Cases Ts-Appl, Ts-Pair, and Ts-Tag Straightforward application of the induction hypothesis.

Case Ts-Match We have

$$\begin{array}{l}\Gamma \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t\\ \Gamma \vdash_s e_0\colon t_0 \qquad t_0 \le \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor\\ \forall i \in I \text{ s.t. } t_i \not\simeq \mathbb{0}.\ \ \Gamma, \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_s e_i\colon t'_i \qquad t = \bigvee_{t_i \not\simeq \mathbb{0}} t'_i\end{array}$$

and must show $\Gamma\theta \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t\theta$.

We prove it by establishing, for some types $\hat{t_0}$ and $\hat{t_i}$, $\hat{t'_i}$ for each i, that

$$\begin{array}{l}\Gamma\theta \vdash_s e_0\colon \hat{t_0} \qquad \hat{t_0} \le \bigvee_{i\in I} \lceil p_i \rfloor \qquad \hat{t_i} = (\hat{t_0} \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor\\ \forall i \in I \text{ s.t. } \hat{t_i} \not\simeq \mathbb{0}.\ \ \Gamma\theta, \mathit{gen}_{\Gamma\theta}(\hat{t_i} /\!/ p_i) \vdash_s e_i\colon \hat{t'_i} \qquad \bigvee_{\hat{t_i} \not\simeq \mathbb{0}} \hat{t'_i} \le t\theta\ .\end{array}$$

Let $A = \{\alpha_1, \ldots, \alpha_n\} = \mathit{var}(t_0) \setminus \mathit{var}(\Gamma)$. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \sharp \theta, \Gamma$. Let $\theta_0 = [{}^{\beta_1}/_{\alpha_1}, \ldots, {}^{\beta_n}/_{\alpha_n}]$ and $\theta' = \theta \circ \theta_0$.

By the induction hypothesis, using θ', we derive $\Gamma\theta' \vdash_s e_0\colon t_0\theta'$, which is $\Gamma\theta \vdash_s e_0\colon t_0\theta'$ (since no α_i is free in Γ, we have $\Gamma\theta' = \Gamma\theta$). We take $\hat{t_0} = t_0\theta'$: note that the exhaustiveness condition is satisfied because substitutions preserves subtyping (and all accepted types of patterns are closed). We have $\hat{t_i} = t_i\theta'$ for all i.

For all branches, note that $\hat{t_i} \not\simeq \mathbb{0}$ implies $t_i \not\simeq \mathbb{0}$. In that case we have $\Gamma, \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_s e_i\colon t'_i$ and, by the induction hypothesis using θ, we can derive $\Gamma\theta, (\mathit{gen}_\Gamma(t_i /\!/ p_i))\theta \vdash_s e_i\colon t'_i\theta$.

We show below $\Gamma\theta, \mathit{gen}_{\Gamma\theta}(t_i\theta' /\!/ p_i) \simeq \Gamma\theta, (\mathit{gen}_\Gamma(t_i /\!/ p_i))\theta$. Since $\mathit{var}(\mathit{gen}_{\Gamma\theta}(t_i\theta' /\!/ p_i)) \subseteq \mathit{var}(\Gamma\theta)$, we have $\mathit{var}(\Gamma\theta, \mathit{gen}_{\Gamma\theta}(t_i\theta' /\!/ p_i)) \subseteq \mathit{var}(\Gamma\theta, (\mathit{gen}_\Gamma(t_i /\!/ p_i))\theta)$. Hence, by Lemma A.20, we derive $\Gamma\theta, \mathit{gen}_{\Gamma\theta}(t_i\theta' /\!/ p_i) \vdash_s e_i\colon t'_i\theta$. We take $\hat{t'_i} = t'_i\theta$.

Finally, we have $\bigvee_{\hat{t}_i \not\leq \mathbb{0}} t_i'\theta \leq \bigvee_{t_i \not\leq \mathbb{0}} t_i'\theta$, because the left union has fewer summands, since $\hat{t}_i \not\leq \mathbb{0}$ implies $t_i \not\leq \mathbb{0}$.

Proof that $\Gamma\theta, \mathit{gen}_{\Gamma\theta}(t_i\theta' /\!/ p_i) \sqsubseteq \Gamma\theta, (\mathit{gen}_\Gamma(t_i /\!/ p_i))\theta$ Recall that $\Gamma\theta = \Gamma\theta'$. We prove, for all $x \in \mathit{capt}(p_i)$, that $\mathit{gen}_{\Gamma\theta'}((t_i\theta' /\!/ p_i)(x)) \sqsubseteq (\mathit{gen}_\Gamma((t_i /\!/ p_i)(x)))\theta$. Let $t_x = (t_i /\!/ p_i)(x)$ and $t_x' = (t_i\theta' /\!/ p_i)(x)$.

We have $\mathit{gen}_\Gamma(t_x) = \forall A'.\, t_x$, where $A' = \mathit{var}(t_x) \setminus \mathit{var}(\Gamma)$. Since $\mathit{var}(t_x) \subseteq \mathit{var}(t_i) = \mathit{var}(t_0)$, $A' \subseteq A$. Let $J = \{\, j \mid \alpha_j \in A' \,\}$; thus $J \subseteq \{1, \ldots, n\}$ and $A' = A|_J = \{\, \alpha_j \mid j \in J \,\}$. Let $B|_J = \{\, \beta_j \mid j \in J \,\}$. We have $\mathit{gen}_\Gamma(t_x) = \forall B|_J.\, t_x\theta_0$ by α-renaming (we are substituting also the α_i such that $i \notin J$, but this is makes no difference since they are not in t_x). Thus, $(\mathit{gen}_\Gamma(t_x))\theta = \forall B|_J.\, t_x\theta_0\theta = \forall B|_J.\, t_x\theta'$, since $B \sharp \theta$.

An instance of $(\mathit{gen}_\Gamma(t_x))\theta$ is then a type $t_x\theta'\theta_x$, with $\mathit{dom}(\theta_x) \subseteq B|_J$. We must find an instance of $\mathit{gen}_{\Gamma\theta'}(t_x')$ which is a subtype of it. Let θ_x' be the restriction of θ_x to variables in $\mathit{var}(t_x') \setminus \mathit{var}(\Gamma\theta')$. Then $t_x'\theta_x' = (t_i\theta' /\!/ p_i)(x)\theta_x'$ is a valid instance of $\mathit{gen}_{\Gamma\theta'}(t_x')$. We prove $(t_i\theta' /\!/ p_i)(x)\theta_x' \leq t_x\theta'\theta_x = (t_i /\!/ p_i)(x)\theta'\theta_x$.

We have $(t_i\theta' /\!/ p_i)(x)\theta_x' = (t_i\theta' /\!/ p_i)(x)\theta_x$: the two substitutions only differ on variables in $B|_J \setminus \mathit{var}((t_i\theta' /\!/ p_i)(x))$ (variables which do not appear in the type at all) and on variables in $B|_J \cap \mathit{var}(\Gamma\theta')$ (which is the empty set, since $\Gamma\theta' = \Gamma\theta$ and $B \sharp \Gamma, \theta$). Then, by Lemma A.18 we have $(t_i\theta' /\!/ p_i)(x) \leq (t_i /\!/ p_i)(x)\theta'$ and hence $(t_i\theta' /\!/ p_i)(x)\theta_x \leq (t_i /\!/ p_i)(x)\theta'\theta_x$.

Case Ts-Subsum The conclusion follows from the induction hypothesis since substitutions preserve subtyping. □

LEMMA A.22: Expression substitution *Let $x_1, \ldots, x_n$ be distinct variables and $v_1, \ldots, v_n$ values. Let $\Gamma' = \{x_1 : s_1, \ldots, x_n : s_n\}$ and $\varsigma = [v_1/x_1, \ldots, v_n/x_n]$.*
If $\Gamma, \Gamma' \vdash_s e : t$ and, for all $k \in \{1, \ldots, n\}$ and for all $t_k \in \mathit{inst}(s_k)$, $\Gamma \vdash_s v_k : t_k$, then $\Gamma \vdash_s e\varsigma : t$.

Proof By induction on the derivation of $\Gamma, \Gamma' \vdash_s e : t$. We reason by cases on the last applied rule.

Case Ts-Var We have

$$\Gamma, \Gamma' \vdash_s x : t \qquad t \in \mathit{inst}((\Gamma, \Gamma')(x))\,.$$

Either $x = x_k$ for some k or not. In the latter case, $x\varsigma = x$, $x \notin \mathit{dom}(\Gamma')$ and hence $(\Gamma, \Gamma')(x) = \Gamma(x)$. Then, since $t \in \mathit{inst}((\Gamma, \Gamma')(x))$, $t \in \mathit{inst}(\Gamma(x))$ and we can apply *Ts-Var*.

If $x = x_k$, then $(\Gamma, \Gamma')(x) = \Gamma'(x) = s_k$. We must then prove $\Gamma \vdash_s v_k : t$, which we know by hypothesis since $t \in \mathit{inst}(s_k)$.

Case Ts-Const Straightforward.

Case Ts-Abstr We have

$$\Gamma, \Gamma' \vdash_s \lambda x.e_1 : t_1 \to t_2 \qquad \Gamma, \Gamma', \{x : t_1\} \vdash_s e_1 : t_2\,.$$

By α-renaming we can assume $x \notin \mathit{dom}(\Gamma')$; then $(\lambda x.e_1)\varsigma = \lambda x.(e_1\varsigma)$ and $\Gamma, \Gamma', \{x : t_1\} = \Gamma, \{x : t_1\}, \Gamma'$. Therefore we have $\Gamma, \{x : t_1\}, \Gamma' \vdash_s e_1 : t_2$ and hence $\Gamma, \{x : t_1\} \vdash_s e_1 : t_2$ by the induction hypothesis. We apply *Ts-Abstr* to conclude.

Cases Ts-Appl, Ts-Pair, Ts-Tag, and Ts-Subsum Straightforward application of the induction hypothesis.

Case Ts-Match We have

$$\Gamma, \Gamma' \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t$$
$$\Gamma, \Gamma' \vdash_s e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor$$
$$\forall\, i \in I \text{ s.t. } t_i \not\leq \mathbb{0}.\ \ \Gamma, \Gamma', \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i) \vdash_s e_i\colon t'_i \qquad t = \bigvee_{t_i \not\leq \mathbb{0}} t'_i\ .$$

We assume by α-renaming that no capture variable of any pattern is in the domain of Γ'. Then, $(\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I})\varsigma = \mathsf{match}\ e_0\varsigma\ \mathsf{with}\ (p_i \to e_i\varsigma)_{i\in I}$ and $\Gamma, \Gamma', \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i) = \Gamma, \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i), \Gamma'$ for any i.

By the induction hypothesis, we derive $\Gamma \vdash_s e_0\varsigma\colon t_0$ and $\Gamma, \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i) \vdash_s e_i\colon t'_i$ for all i. From the latter, we prove $\Gamma, \mathit{gen}_{\Gamma}(t_i /\!/ p_i) \vdash_s e_i\colon t'_i$ by weakening (Lemma A.20): we have $\mathit{gen}_{\Gamma}(t_i /\!/ p_i) \sqsubseteq \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i)$ by Lemma A.19 – since $\mathit{var}(\Gamma) \subseteq \mathit{var}(\Gamma, \Gamma')$ – and clearly we have $\mathit{var}(\Gamma, \mathit{gen}_{\Gamma}(t_i /\!/ p_i)) \subseteq \mathit{var}(\Gamma, \mathit{gen}_{\Gamma,\Gamma'}(t_i /\!/ p_i))$ since $\mathit{var}(\mathit{gen}_{\Gamma}(t_i /\!/ p_i)) \subseteq \mathit{var}(\Gamma)$. □

Type soundness

THEOREM A.23: Progress *Let e be a well-typed, closed expression. Then, either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof By hypothesis we have $\varnothing \vdash_s e\colon t$. The proof is by induction on its derivation; we reason by cases on the last applied rule.

Case Ts-Var This case does not occur because variables are not closed.

Case Ts-Const In this case e is a constant c and therefore a value.

Case Ts-Abstr In this case e is an abstraction $\lambda x.e_1$. Since it is also closed, it is a value.

Case Ts-Appl We have

$$\varnothing \vdash_s e_1\ e_2\colon t \qquad \varnothing \vdash_s e_1\colon t' \to t \qquad \varnothing \vdash_s e_2\colon t'\ .$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e'_1$, then $e_1\ e_2 \rightsquigarrow e'_1\ e_2$. If e_1 is a value and $e_2 \rightsquigarrow e'_2$, then $e_1\ e_2 \rightsquigarrow e_1\ e'_2$.

If both are values then, by Lemma A.12, e_1 has the form $\lambda x.e_3$ for some e_3. Then, we can apply *R-Appl* and $e_1\ e_2 \rightsquigarrow e_3[e_2/x]$.

Case Ts-Pair We have

$$\varnothing \vdash_s (e_1, e_2)\colon t_1 \times t_2 \qquad \varnothing \vdash_s e_1\colon t_1 \qquad \varnothing \vdash_s e_2\colon t_2\ .$$

By the induction hypothesis, each of e_1 and e_2 either is a value or may reduce. If $e_1 \rightsquigarrow e'_1$, then $(e_1, e_2) \rightsquigarrow (e'_1, e_2)$. If e_1 is a value and $e_2 \rightsquigarrow e'_2$, then $(e_1, e_2) \rightsquigarrow (e_1, e'_2)$. If both are values, then (e_1, e_2) is also a value.

Case Ts-Tag We have

$$\varnothing \vdash_s \text{`}\mathit{tag}(e_1)\colon \text{`}\mathit{tag}(t_1) \qquad \varnothing \vdash_s e_1\colon t_1\ .$$

Analogously to the previous case, by the induction hypothesis we have that either e_1 is a value or $e_1 \rightsquigarrow e'_1$. In the former case, $\text{`}\mathit{tag}(e_1)$ is a value as well. In the latter, we have $\text{`}\mathit{tag}(e_1) \rightsquigarrow \text{`}\mathit{tag}(e'_1)$.

Case Ts-Match We have

$$\varnothing \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \qquad \varnothing \vdash_{\mathrm{s}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \ .$$

By the inductive hypothesis, either e_0 is a value or it may reduce. In the latter case, if $e_0 \rightsquigarrow e_0'$, then $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow \mathsf{match}\ e_0'\ \mathsf{with}\ (p_i \to e_i)_{i\in I}$.

If e_0 is a value, on the other hand, the expression may reduce by application of *R-Match*. Since $t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor$, $\varnothing \vdash_{\mathrm{s}} e_0\colon \bigvee_{i\in I} \lceil p_i \rfloor$ holds by subsumption. Hence, since e_0 is a value, $\varnothing \vdash_{\mathrm{s}} e_0\colon \lceil p_i \rfloor$ holds for at least one i (by Lemma A.13); for each such i we have $e_0/p_i = \varsigma_i$ (by Lemma A.15). Let j be the least of these i; then $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow e_j\varsigma_j$.

Case Ts-Subsum Straightforward application of the induction hypothesis. □

THEOREM A.24: Subject reduction *Let e be an expression and t a type such that $\Gamma \vdash_{\mathrm{s}} e\colon t$. If $e \rightsquigarrow e'$, then $\Gamma \vdash_{\mathrm{s}} e'\colon t$.*

Proof By induction on the derivation of $\Gamma \vdash_{\mathrm{s}} e\colon t$. We reason by cases on the last applied rule.

Cases Ts-Var, Ts-Const, and Ts-Abstr These cases may not occur: variables, constants, and abstractions never reduce.

Case Ts-Appl We have

$$\Gamma \vdash_{\mathrm{s}} e_1\ e_2\colon t \qquad \Gamma \vdash_{\mathrm{s}} e_1\colon t' \to t \qquad \Gamma \vdash_{\mathrm{s}} e_2\colon t' \ .$$

$e_1\ e_2 \rightsquigarrow e'$ occurs in any of three ways: *i*) $e_1 \rightsquigarrow e_1'$ and $e' = e_1'\ e_2$; *ii*) e_1 is a value, $e_2 \rightsquigarrow e_2'$ and $e' = e_1\ e_2'$; *iii*) both e_1 and e_2 are values, e_1 is of the form $\lambda x.e_3$, and $e' = e_3[e_2/x]$.

In the first case, we derive by the induction hypothesis that $\Gamma \vdash_{\mathrm{s}} e_1'\colon t' \to t$ and conclude by applying *Ts-Appl* again. The second case is analogous.

In the third case, we know by Lemma A.12 that $\Gamma, \{x\colon t'\} \vdash_{\mathrm{s}} e_3\colon t$. We also know that e_2 is a value such that $\Gamma \vdash_{\mathrm{s}} e_2\colon t'$. Then, by Lemma A.22, $\Gamma \vdash_{\mathrm{s}} e_3[e_2/x]\colon t$.

Case Ts-Pair We have

$$\Gamma \vdash_{\mathrm{s}} (e_1, e_2)\colon t_1 \times t_2 \qquad \Gamma \vdash_{\mathrm{s}} e_1\colon t_1 \qquad \Gamma \vdash_{\mathrm{s}} e_2\colon t_2 \ .$$

$(e_1, e_2) \rightsquigarrow e'$ occurs either because $e_1 \rightsquigarrow e_1'$ and $e' = (e_1', e_2)$, or because e_1 is a value, $e_2 \rightsquigarrow e_2'$, and $e' = (e_1, e_2')$. In either case, the induction hypothesis allows us to derive that the type of the component that reduces is preserved; therefore, we can apply *Ts-Pair* again to conclude.

Case Ts-Tag Analogously to the previous case, a variant expression only reduces if its argument does, so we apply the induction hypothesis and *Ts-Tag* to conclude.

Case Ts-Match We have

$$\begin{array}{l}\Gamma \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \\ \Gamma \vdash_{\mathrm{s}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \forall\, i \in I \text{ s.t. } t_i \not\leq \mathbb{0}.\ \ \Gamma, \mathit{gen}_\Gamma(t_i /\!/ p_i) \vdash_{\mathrm{s}} e_i\colon t_i' \qquad t = \bigvee_{t_i \not\leq \mathbb{0}} t_i' \ .\end{array}$$

The reduction $\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I} \rightsquigarrow e'$ occurs either because $e_0 \rightsquigarrow e_0'$ and $e' = \mathsf{match}\ e_0'\ \mathsf{with}\ (p_i \to e_i)_{i\in I}$ or because e_0 is a value and $e' = e_j\varsigma$, where $e_0/p_j = \varsigma$ and, for all $i < j$, $e_0/p_i = \Omega$. In the former case, we apply the induction hypothesis and conclude by *Ts-Match*.

$$\textit{R-Fix}\ \frac{}{\Upsilon\,(\lambda x.e)\ \rightsquigarrow\ e\left[\Upsilon\,(\lambda x.e)/x\right]} \qquad \textit{Tk-Fix}\ \frac{K;\Gamma \vdash_{\kappa} e\colon \tau \to \tau}{K;\Gamma \vdash_{\kappa} \Upsilon\, e\colon \tau} \qquad \textit{Ts-Fix}\ \frac{\Gamma \vdash_{s} e\colon t \to t}{\Gamma \vdash_{s} \Upsilon\, e\colon t}$$

FIGURE A.1 Reduction and typing rules for the fixed-point combinator.

In the latter case, ς is a substitution from the capture variables of p_j to values. We can derive

$$\Gamma \vdash_{s} e_0\colon \lbag p_j \rbag \qquad \forall\, i < j.\ \ \Gamma \vdash_{s} e_0\colon \neg \lbag p_i \rbag$$

by Lemma A.15 and thence $\Gamma \vdash_{s} e_0\colon t_j$ by Lemma A.14. Therefore, by Lemma A.17, we have that, for all $x \in \mathit{capt}(p_j)$, $\Gamma \vdash_{s} x\varsigma\colon (t_j /\!/ p_j)(x)$. Let $\Gamma' = t_j /\!/ p_j$.

We show that, additionally, $\Gamma \vdash_{s} x\varsigma\colon t_x$ holds for every $t_x \in \mathit{inst}(\mathit{gen}_{\Gamma}(\Gamma'(x)))$. Every such t_x is equal to $\Gamma'(x)\theta$ for a θ such that $\mathit{dom}(\theta) \subseteq \mathit{var}(\Gamma'(x)) \setminus \mathit{var}(\Gamma)$. Then, $\Gamma \vdash_{s} x\varsigma\colon \Gamma'(x)\theta$ holds by Lemma A.21, since $\Gamma\theta = \Gamma$ (the substitution does not change any free variable of Γ).

From $\Gamma, \mathit{gen}_{\Gamma}(\Gamma') \vdash_{s} e_j\colon t'_j$ and from the fact that we have $\Gamma \vdash_{s} x\varsigma\colon t_x$ for all $x \in \mathit{capt}(p_j)$ and all $t_x \in \mathit{inst}(\mathit{gen}_{\Gamma}(\Gamma'(x)))$, we derive $\Gamma \vdash_{s} e_j\varsigma\colon t'_j$ by Lemma A.22 and then conclude by subsumption.

Case Ts-Subsum Straightforward application of the induction hypothesis. □

COROLLARY A.25: Type soundness *Let e be a well-typed, closed expression, that is, such that $\varnothing \vdash_{s} e\colon t$ holds for some t. Then, either e diverges or it reduces to a value v such that $\varnothing \vdash_{s} v\colon t$.*

Proof Consequence of Theorem A.23 and Theorem A.24. □

Completeness with respect to VariantsK

ADDING RECURSIVE FUNCTIONS To prove the result of this section, we extend the *Variants* calculus and its type systems with the possibility of defining recursive functions by a fixed-point combinator. This simplifies the proof because it allows us to assume that all arrow types are inhabited.

The extension of the calculus is done by adding a new production $\Upsilon\ e$ to the grammar defining expressions, a new production $\Upsilon\ E$ to the grammar of evaluation contexts, and the new reduction rule *R-Fix* in Figure A.1. We extend *VariantsK* and *VariantsS* with the addition, respectively, of the rules *Tk-Fix* and *Ts-Fix* in Figure A.1.

LEMMA A.26 *For any k-type τ in a non-recursive kinding environment K, we have $\mathit{var}(\llbracket \tau \rrbracket_K) \subseteq \mathit{var}_K(\tau)$. Likewise, for any k-scheme σ and k-type environment Γ, we have $\mathit{var}(\llbracket \sigma \rrbracket_K) \subseteq \mathit{var}_K(\sigma)$ and $\mathit{var}(\llbracket \Gamma \rrbracket_K) \subseteq \mathit{var}_K(\Gamma)$.*

Proof The translation does not introduce new variables, therefore we can show $\mathit{var}(\llbracket \tau \rrbracket_K) \subseteq \mathit{var}_K(\tau)$ by induction on $w(\tau, K)$. We extend this straightforwardly to type schemes and environments. □

LEMMA A.27 *Let p be a pattern and $t \leq \lbag p \rbag$ an s-type. If $K \vdash p\colon \tau \Rightarrow \Gamma$ and $t \leq \llbracket \tau \rrbracket_K$, then, for all $x \in \mathit{capt}(p)$, $(t /\!/ p)(x) \leq \llbracket \Gamma(x) \rrbracket_K$.*

Proof By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathit{capt}(p) = \varnothing$.

Case $p = x$ We have

$$K \vdash p\colon \tau \Rightarrow \{x\colon \tau\} \qquad t/\!/x = \{x\colon t\}$$

and must prove $\{x\colon t\}(x) \leq [\![\{x\colon \tau\}(x)]\!]_K$, that is, $t \leq [\![\tau]\!]_K$, which is true by hypothesis.

Case $p = (p_1, p_2)$ We have

$$K \vdash p\colon \tau_1 \times \tau_2 \Rightarrow \Gamma \qquad \Gamma = \Gamma_1 \cup \Gamma_2 \qquad \forall i.\ K \vdash p_i\colon \tau_i \Rightarrow \Gamma_i$$
$$t/\!/p = \pi_1(t)/\!/p_1 \cup \pi_2(t)/\!/p_2\,.$$

Since $t \leq [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K$, by Property 4.10 we have $\pi_i(t) \leq [\![\tau_i]\!]_K$. Likewise, $\pi_i(t) \leq \lfloor p_i \rfloor$. We apply the induction hypothesis to conclude.

Case $p = \backprime tag(p_1)$ We have

$$K \vdash p\colon \alpha \Rightarrow \Gamma \qquad K \vdash p_1\colon \tau_1 \Rightarrow \Gamma \qquad \alpha :: (L, U, T) \in K \qquad \backprime tag \in U \Rightarrow \backprime tag\colon \tau_1 \in T$$
$$t/\!/p = \pi_{\backprime tag}(t)/\!/p_1\,.$$

Since $t \leq \lfloor \backprime tag(p_1) \rfloor = \backprime tag(\lfloor p_1 \rfloor)$, by Property 4.11 we have $\pi_{\backprime tag}(t) \leq \lfloor p_1 \rfloor$. We next prove $\pi_{\backprime tag}(t) \leq [\![\tau_1]\!]_K$, which allows us to apply the induction hypothesis and conclude.

The translation of α is $[\![\alpha]\!]_K = (\mathit{low}_K(L, T) \vee \alpha) \wedge \mathit{upp}_K(U, T)$. We have $t \leq [\![\alpha]\!]_K$ and hence $t \leq \mathit{upp}_K(U, T)$. Since $t \leq \backprime tag(\mathbb{1})$, $t \leq \mathit{upp}_K(U, T) \wedge \backprime tag(\mathbb{1})$. We distribute the intersection over the summands of $\mathit{upp}_K(U, T)$, which is a union.

If $\backprime tag \notin U$ (in which case $U \neq \mathcal{L}$), then all summands have the form $\backprime tag_1(\tau')$ and for each $\backprime tag_1$ we have $\backprime tag_1 \neq \backprime tag$: hence, the intersection is empty and thus we have $t \leq \mathbb{0} \simeq \backprime tag(\mathbb{0})$. Then $\pi_{\backprime tag}(t) \leq \mathbb{0} \leq [\![\tau_1]\!]_K$.

If $\backprime tag \in U$, then necessarily $\backprime tag \in \mathit{dom}(T)$ holds as well. In that case the intersection $\mathit{upp}_K(U, T) \wedge \backprime tag(\mathbb{1})$ is equivalent to $\backprime tag(\bigwedge_{\backprime tag:\tau' \in T} [\![\tau']\!]_K)$. Hence $t \leq \backprime tag(\bigwedge_{\backprime tag:\tau' \in T} [\![\tau']\!]_K)$ and $\pi_{\backprime tag}(t) \leq \bigwedge_{\backprime tag:\tau' \in T} [\![\tau']\!]_K$. Since $\backprime tag\colon \tau_1 \in T$, $\bigwedge_{\backprime tag:\tau' \in T} [\![\tau']\!]_K \leq [\![\tau_1]\!]_K$, from which follows $\pi_{\backprime tag}(t) \leq [\![\tau_1]\!]_K$.

Case $p = p_1 \& p_2$ We directly apply the induction hypothesis to both sub-patterns and conclude.

Case $p = p_1 | p_2$ We have

$$K \vdash p\colon \tau \Rightarrow \Gamma \qquad \forall i.\ K \vdash p_i\colon \tau \Rightarrow \Gamma$$
$$t/\!/p = (t \wedge \lfloor p_1 \rfloor)/\!/p_1 \vee (t \setminus \lfloor p_1 \rfloor)/\!/p_2\,.$$

Since $t \wedge \lfloor p_1 \rfloor$ and $t \setminus \lfloor p_1 \rfloor$ are subtypes of t, they are also subtypes of $[\![\tau]\!]_K$. We can apply the induction hypothesis and, for each x, derive both that $(t \wedge \lfloor p_1 \rfloor/\!/p_1)(x) \leq [\![\Gamma(x)]\!]_K$ and that $(t \setminus \lfloor p_1 \rfloor/\!/p_2)(x) \leq [\![\Gamma(x)]\!]_K$. Hence, $(t/\!/p)(x) \leq [\![\Gamma(x)]\!]_K$. □

LEMMA A.28: Translation of type substitutions *Let K, K' be two non-recursive kinding environments such that $\mathit{dom}(K') \cap (\mathit{dom}(K) \cup \mathit{var}_\varnothing(K)) = \varnothing$. Let θ be a k-type substitution such that $\mathit{dom}(\theta) \subseteq \mathit{dom}(K')$ and $K, K' \vdash \theta\colon K$.*

Let θ' be the s-type substitution defined as $[\, [\![\alpha\theta]\!]_K/\alpha \mid \alpha \in \mathit{dom}(K') \,]$. For every k-type τ, we have $[\![\tau]\!]_{K,K'}\theta' \simeq [\![\tau\theta]\!]_K$.

Proof By complete induction on $w(\tau, (K, K'))$. We proceed by cases on τ and assume that the lemma holds for all τ' such that $w(\tau', (K, K')) < w(\tau, (K, K'))$.

Case $\tau = \alpha$, with $\alpha :: \bullet \in K, K'$ We have $[\![\alpha]\!]_{K,K'} = \alpha$, hence $[\![\alpha]\!]_{K,K'}\theta' = \alpha\theta'$. Either $\alpha \in \mathit{dom}(K)$ or $\alpha \in \mathit{dom}(K')$ (the domains are disjoint). In the former case, $\alpha\theta = \alpha$ and $\alpha\theta' = \alpha$. Thus we have $[\![\alpha]\!]_{K,K'}\theta' = \alpha = [\![\alpha\theta]\!]_K$. In the latter, $\alpha\theta' = [\![\alpha\theta]\!]_K$ holds by definition of θ'.

Case $\tau = \alpha$, with $\alpha :: (L, U, T) \in K$ and $\alpha \notin \mathit{dom}(K')$ We have $[\![\alpha]\!]_{K,K'} = [\![\alpha]\!]_K$ because no variable in the kind of α is in $\mathit{dom}(K')$. For the same reason, since the translation does not add variables, $[\![\alpha]\!]_{K,K'}\theta' = [\![\alpha]\!]_K$. Additionally, $\alpha\theta = \alpha$, so also $[\![\alpha\theta]\!]_K = [\![\alpha]\!]_K$.

Case $\tau = \alpha$, with $\alpha :: (L', U', T') \in K'$ Because $K, K' \vdash \theta : K$, we know that $\alpha\theta$ is some variable β such that $\beta :: (L, U, T) \in K$ and $(L, U, T) \vDash (L', U', T'\theta)$.

We have

$$[\![\alpha\theta]\!]_K \;=\; [\![\beta]\!]_K \;=\; (\mathit{low}_K(L, T) \vee \beta) \wedge \mathit{upp}_K(U, T)$$

and

$$\begin{aligned}
&[\![\alpha]\!]_{K,K'}\theta' \;=\; \big((\mathit{low}_{K,K'}(L', T') \vee \alpha) \wedge \mathit{upp}_{K,K'}(U', T')\big)\theta' \;= \\
&(\mathit{low}_{K,K'}(L', T')\theta' \vee \alpha\theta') \wedge \mathit{upp}_{K,K'}(U', T')\theta' \;= \\
&\Big(\mathit{low}_{K,K'}(L', T')\theta' \vee \big((\mathit{low}_K(L, T) \vee \beta) \wedge \mathit{upp}_K(U, T)\big)\Big) \wedge \mathit{upp}_{K,K'}(U', T')\theta' \,.
\end{aligned}$$

Let us define

$$\begin{aligned}
&l = \mathit{low}_K(L, T) \qquad u = \mathit{upp}_K(U, T) \\
&l' = \mathit{low}_{K,K'}(L', T')\theta' \qquad u' = \mathit{upp}_{K,K'}(U', T')\theta'
\end{aligned}$$

and assume that the following hold (we prove them below):

$$l \le u \qquad l' \le u' \qquad l' \le l \qquad u \le u' \,.$$

Then we have also $l' \le u$ by transitivity. Whenever $t \le t'$, we have $t \wedge t' \simeq t$ and $t \vee t' \simeq t'$.

Thus we have the following equivalences:

$$\begin{aligned}
[\![\alpha]\!]_{K,K'}\theta' \;&=\; (l' \vee ((l \vee \beta) \wedge u)) \wedge u' \\
&\simeq\; (l' \wedge u') \vee ((l \vee \beta) \wedge u \wedge u') && \text{distributivity} \\
&\simeq\; l' \vee ((l \vee \beta) \wedge u) && l' \le u' \text{ and } u \le u' \\
&\simeq\; (l' \vee l \vee \beta) \wedge (l' \vee u) && \text{distributivity} \\
&\simeq\; (l \vee \beta) \wedge u && l' \le l \text{ and } l' \le u
\end{aligned}$$

by which we conclude.

We now prove our four assumptions. The first, $l \le u$, holds because $L \subseteq U$ and $L \subseteq \mathit{dom}(T)$: hence each branch of l appears in u as well. The second is analogous.

For the other assumptions, note that $[\![\tau']\!]_{K,K'}\theta' \simeq [\![\tau'\theta]\!]_K$ holds for all τ' in the range of T'. To prove $l' \le l$, note that $L' \subseteq L$ and $T'\theta \subseteq T$. In l', we distribute the application of θ' over all the summands of the union and inside all variant type constructors. Then, we show $\text{`}tag(\bigwedge_{\text{`}tag:\tau' \in T'} [\![\tau']\!]_{K,K'}\theta') \le l$ for each $\text{`}tag \in L'$. We have $\text{`}tag(\bigwedge_{\text{`}tag:\tau' \in T'} [\![\tau']\!]_{K,K'}\theta') \simeq \text{`}tag(\bigwedge_{\text{`}tag:\tau' \in T'} [\![\tau'\theta]\!]_K) = \text{`}tag(\bigwedge_{\text{`}tag:\tau'\theta \in T'\theta} [\![\tau'\theta]\!]_K)$. Since $L' \subseteq L$, there is a summand of l with

the same tag. Since $\backprime tag$ is in the lower bound, it has a single type in both T and T' and, since $T'\theta \subseteq T$, the type it has in T must be $\tau'\theta$.

To prove $u \leq u'$, note that $U \subseteq U'$. If $U = \mathcal{L}$, then $U' = \mathcal{L}$. Then both u and u' are unions of two types: the union of tags mentioned respectively in T and T' and the rest. For each $\backprime tag$, if $\backprime tag \notin dom(T)$, then $\backprime tag \notin dom(T')$, in which case both u and u' admit it with any argument type. If $\backprime tag \in dom(T)$, either $\backprime tag \in dom(T')$ or not. In the former case, u admits a smaller argument type than u' because $T'\theta \subseteq T$. The same occurs in the latter case, since u' admits $\backprime tag$ with any argument type.

If $U \neq \mathcal{L}$, then U' could be $\mathcal{L}$ or not. In either case we can prove, for each $\backprime tag \in U$, that u' admits $\backprime tag$ with a larger argument type than u does.

Case $\tau = b$ Straightforward, since a basic type is translated into itself and is never affected by substitutions.

Case $\tau = \tau_1 \to \tau_2$ By the induction hypothesis we have $[\![\tau_i]\!]_{K,K'}\theta' \simeq [\![\tau_i\theta]\!]_K$ for both i. Then

$$[\![\tau_1 \to \tau_2]\!]_{K,K'}\theta' = ([\![\tau_1]\!]_{K,K'}\theta') \to ([\![\tau_2]\!]_{K,K'}\theta') \simeq$$
$$([\![\tau_1\theta]\!]_K) \to ([\![\tau_2\theta]\!]_K) = [\![(\tau_1\theta) \to (\tau_2\theta)]\!]_K = [\![(\tau_1 \to \tau_2)\theta]\!]_K .$$

Case $\tau = \tau_1 \times \tau_2$ Analogous to the previous case. □

LEMMA A.29 *If* $\varnothing \vdash_s v : [\![\tau]\!]_K$*, then there exists a value* v' *such that* $K; \varnothing \vdash_\kappa v' : \tau$ *and, for every pattern* p, $v/p = \Omega \iff v'/p = \Omega$.

Proof By structural induction on v.

Note that values are always typed by an application of the typing rule corresponding to their form (*Ts-Const*, *Ts-Abstr*, *Ts-Pair*, or *Ts-Tag*) to appropriate premises, possibly followed by applications of *Ts-Subsum*. Hence, if $\varnothing \vdash_s v : t$, there is a type $t' \leq t$ such that $\varnothing \vdash_s v : t'$ and that the last typing rule used to derive $\varnothing \vdash_s v : t'$ is one of the four above, given by the form of v.

Case $v = c$ We have $[\![\tau]\!]_K \geq c$. Hence $\tau = b_c$, as the translation of any other τ is disjoint from c. Then we can take $v' = v$.

Case $v = (v_1, v_2)$ We have $[\![\tau]\!]_K \geq t_1 \times t_2$ for some t_1 and t_2. Hence $\tau = \tau_1 \times \tau_2$: any other τ would translate to a type disjoint from all products. Therefore $\varnothing \vdash_s v : [\![\tau_1]\!]_K \times [\![\tau_2]\!]_K$. By Lemma A.12 we have $\varnothing \vdash_s v_i : [\![\tau_i]\!]_K$ for both i; then by the induction hypothesis we find v_i' for both i and let $v' = (v_1', v_2')$.

Case $v = \backprime tag(v_1)$ We have $[\![\tau]\!]_K \geq \backprime tag(t_1)$ and $\varnothing \vdash_s v : \backprime tag(t_1)$ for some $t_1 \not\leq \mathbb{0}$ (since t_1 types the value v_1). Therefore, by the same reasoning as above, $\tau = \alpha$ with $\alpha :: (L, U, T) \in K$. Since $[\![\tau]\!]_K \geq \backprime tag(t_1)$, we have $\backprime tag \in L$ and therefore $\backprime tag : \tau_1 \in T$ for some τ_1 such that $t_1 \leq [\![\tau_1]\!]_K$. Then we have $\varnothing \vdash_s v_1 : [\![\tau_1]\!]_K$; we may apply the induction hypothesis to find a value v_1' and let $v' = \backprime tag(v_1')$.

Case $v = \lambda x.e$ Note that an abstraction is only accepted by patterns which accept any value, so any two abstractions fail to match exactly the same patterns.

We have $\varnothing \vdash_s v : t_1 \to t_2$ for some $t_1 \to t_2 \leq [\![\tau]\!]_K$. Hence we know τ is of the form $\tau_1 \to \tau_2$; thus we have $\varnothing \vdash_s v : [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K$. We take v' to be the function $\lambda x.\Upsilon\,(\lambda f.\lambda x.f\ x)\ x$, which never terminates and can be assigned any arrow type. □

LEMMA A.30 *Let K be a kinding environment, τ a k-type, and P a set of patterns. If $\tau \preccurlyeq_K P$, then $[\![\tau]\!]_K \leq \bigvee_{p \in P} \lfloor p \rfloor$.*

Proof By contradiction, assume that $\tau \preccurlyeq_K P$ holds but $[\![\tau]\!]_K \not\leq \bigvee_{p \in P} \lfloor p \rfloor$. The latter condition implies that there exists a value v in the interpretation of $[\![\tau]\!]_K$ which is not in the interpretation of $\bigvee_{p \in P} \lfloor p \rfloor$. Because the definition of accepted type is exact with respect to the semantics of pattern matching, we have $v/p = \Omega$ for all $p \in P$. We also have $\varnothing \vdash_{\text{s}} v : [\![\tau]\!]_K$ since v is in the interpretation of that type (typing is complete with respect to the interpretation if we restrict ourselves to translations of k-types).

By Lemma A.29, from v we can build a value v' such that $K; \varnothing \vdash_{\text{k}} v' : \tau$ and, for every pattern p, $v/p = \Omega \iff v'/p = \Omega$. We reach a contradiction, since $\tau \preccurlyeq_K P$ and $K; \varnothing \vdash_{\text{k}} v' : \tau$ imply $v'/p \neq \Omega$ for all $p \in P$, whereas we have $v/p = \Omega$ for all $p \in P$. □

THEOREM A.31: Preservation of typing *Let e be an expression, K a non-recursive kinding environment, Γ a k-type environment, and τ a k-type. If $K; \Gamma \vdash_{\text{k}} e : \tau$, then $[\![\Gamma]\!]_K \vdash_{\text{s}} e : [\![\tau]\!]_K$.*

Proof By induction on the derivation of $K; \Gamma \vdash_{\text{k}} e : \tau$. We reason by cases on the last applied rule.

Case Tk-Var We have

$$K; \Gamma \vdash_{\text{k}} x : \tau \qquad \tau \in \mathit{inst}_K(\Gamma(x)) \qquad \text{hence}$$
$$\Gamma(x) = \forall A.\, K_x \triangleright \tau_x \qquad \tau = \tau_x\theta \qquad \mathit{dom}(\theta) \subseteq A \qquad K, K_x \vdash \theta : K$$

and must show $[\![\Gamma]\!]_K \vdash_{\text{s}} x : [\![\tau]\!]_K$. Since $[\![\Gamma]\!]_K(x) = \forall A.\, [\![\tau_x]\!]_{K,K_x}$, by *Ts-Var* we can derive $[\![\tau_x]\!]_{K,K_x}\theta'$ for any s-type substitution θ' with $\mathit{dom}(\theta') \subseteq A$.

Consider the s-type substitution $\theta' = [\, [\![\alpha\theta]\!]_K/\alpha \mid \alpha \in A \,]$. We have $[\![\tau_x]\!]_{K,K_x}\theta' \simeq [\![\tau_x\theta]\!]_K$ by Lemma A.28 (we can assume the conditions on the domain of K_x to hold by renaming the variables in A). Hence, we derive $[\![\tau_x]\!]_{K,K_x}\theta'$ by *Ts-Var* and then $[\![\tau_x\theta]\!]_K$ by subsumption.

Case Tk-Const We have

$$K; \Gamma \vdash_{\text{k}} c : b_c \qquad [\![b_c]\!]_K = b_c$$

and may derive $[\![\Gamma]\!]_K \vdash_{\text{s}} c : c$ by *Ts-Const* and $[\![\Gamma]\!]_K \vdash_{\text{s}} c : b_c$ by subsumption.

Case Tk-Abstr We have

$$K; \Gamma \vdash_{\text{k}} \lambda x.e_1 : \tau_1 \to \tau_2 \qquad K; \Gamma, \{x : \tau_1\} \vdash_{\text{k}} e_1 : \tau_2 \qquad [\![\tau_1 \to \tau_2]\!]_K = [\![\tau_1]\!]_K \to [\![\tau_2]\!]_K\,.$$

By the induction hypothesis we derive $[\![\Gamma]\!]_K, \{x : [\![\tau_1]\!]_K\} \vdash_{\text{s}} e_1 : [\![\tau_2]\!]_K$, then we apply *Ts-Abstr*.

Cases Tk-Appl, Tk-Pair, and Tk-Fix Straightforward application of the induction hypothesis.

Case Tk-Tag We have

$$K; \Gamma \vdash_{\text{k}} \text{`}tag(e_1) : \alpha \qquad K; \Gamma \vdash_{\text{k}} e_1 : \tau_1 \qquad \alpha :: (L, U, T) \in K \qquad \text{`}tag \in L \qquad \text{`}tag : \tau_1 \in T$$
$$[\![\alpha]\!]_K = (\mathit{low}_K(L, T) \vee \alpha) \wedge \mathit{upp}_K(U, T)\,.$$

We derive $[\![\Gamma]\!]_K \vdash_{\text{s}} e_1 : [\![\tau_1]\!]_K$ by the induction hypothesis, then $[\![\Gamma]\!]_K \vdash_{\text{s}} \text{`}tag(e_1) : \text{`}tag([\![\tau_1]\!]_K)$ by *Ts-Tag*. We show that $\text{`}tag([\![\tau_1]\!]_K) \leq [\![\alpha]\!]_K$ holds: hence, we may derive the supertype by subsumption.

Since $\text{`}tag \in L$ and hence $\text{`}tag \in \mathit{dom}(T)$, both $\mathit{low}_K(L, T)$ and $\mathit{upp}_K(U, T)$ contain a sum-

mand $\backprime tag(\bigwedge_{\backprime tag:\tau' \in T} [\![\tau']\!]_K)$. Since $\backprime tag\colon \tau_1 \in T$ and no other type may be associated to $\backprime tag$, the intersection has a single factor $[\![\tau_1]\!]_K$. Thus we have both $\backprime tag([\![\tau_1]\!]_K) \leq \mathit{low}_K(L, T)$ and $\backprime tag([\![\tau_1]\!]_K) \leq \mathit{upp}_K(U, T)$; hence, $\backprime tag([\![\tau_1]\!]_K) \leq [\![\alpha]\!]_K$.

Case Tk-Match We have

$$K; \Gamma \vdash_\kappa \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \colon \tau$$
$$K; \Gamma \vdash_\kappa e_0 \colon \tau_0 \qquad \tau_0 \lesssim_K \{\, p_i \mid i \in I \,\}$$
$$\forall i \in I.\ \ K \vdash p_i \colon \tau_0 \Rightarrow \Gamma_i \qquad K; \Gamma, \mathit{gen}_{K;\Gamma}(\Gamma_i) \vdash_\kappa e_i \colon \tau$$

and must show

$$[\![\Gamma]\!]_K \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I} \colon [\![\tau]\!]_K$$

which we prove by establishing, for some types t_0 and t_i, t_i' for each i, that

$$[\![\Gamma]\!]_K \vdash_s e_0 \colon t_0 \qquad t_0 \leq \bigvee_{i \in I} \lfloor p_i \rfloor \qquad t_i = (t_0 \setminus \bigvee_{j<i} \lfloor p_j \rfloor) \wedge \lfloor p_i \rfloor$$
$$\forall i \in I \text{ s.t. } t_i \not\leq \mathbb{0}.\ \ [\![\Gamma]\!]_K, \mathit{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \vdash_s e_i \colon t_i' \qquad \bigvee_{t_i \not\leq \mathbb{0}} t_i' \leq [\![\tau]\!]_K\,.$$

and then applying *Ts-Match*, followed by *Ts-Subsum* if necessary.

By the induction hypothesis we derive $[\![\Gamma]\!]_K \vdash_s e_0 \colon [\![\tau_0]\!]_K$ and hence have $t_0 = [\![\tau_0]\!]_K$. By Lemma A.30, we have $t_0 \leq \bigvee_{i \in I} \lfloor p_i \rfloor$. For every branch, $t_i \leq t_0$ and $t_i \leq \lfloor p_i \rfloor$: therefore, we can apply Lemma A.27 and derive that $(t_i /\!/ p_i)(x) \leq [\![\Gamma_i(x)]\!]_K$ holds for every $x \in \mathit{capt}(p_i)$.

For each branch, we derive $[\![\Gamma]\!]_K, [\![\mathit{gen}_{K;\Gamma}(\Gamma_i)]\!]_K \vdash_s e_i \colon [\![\tau]\!]_K$ by the induction hypothesis. We derive $[\![\Gamma]\!]_K, \mathit{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \vdash_s e_i \colon [\![\tau]\!]_K$ by Lemma A.20 by proving $[\![\Gamma]\!]_K, \mathit{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i) \sqsubseteq [\![\Gamma]\!]_K, [\![\mathit{gen}_{K;\Gamma}(\Gamma_i)]\!]_K$ and $\mathit{var}([\![\Gamma]\!]_K, \mathit{gen}_{[\![\Gamma]\!]_K}(t_i /\!/ p_i)) \subseteq \mathit{var}([\![\Gamma]\!]_K, [\![\mathit{gen}_{K;\Gamma}(\Gamma_i)]\!]_K)$. The latter is straightforward. For the former, for each $x \in \mathit{capt}(p_i)$ – say $\Gamma_i(x) = \tau_x$ and $(t_i /\!/ p_i)(x) = t_x$ – we must show $\mathit{gen}_{[\![\Gamma]\!]_K}(t_x) \sqsubseteq [\![\mathit{gen}_{K;\Gamma}(\tau_x)]\!]_K$. This holds because $t_x \leq [\![\tau_x]\!]_K$ and because, by Lemma A.26, $\mathit{var}([\![\Gamma]\!]_K) \subseteq \mathit{var}_K(\Gamma)$.

We can thus choose $t_i' = [\![\tau]\!]_K$ for all branches, satisfying $\bigvee_{t_i \not\leq \mathbb{0}} t_i' \leq [\![\tau]\!]_K$. □

A.3 Reconstruction for set-theoretic types

Constraint generation with explicit variables We give an alternative definition of the two relations $e\colon t \Rightarrow C$ and $t /\!/\!/ p \Rightarrow (\Gamma, C)$ (given by the rules in Figure 5.2 on page 56 and in Figure 5.3 on page 57, respectively) where we keep track explicitly of the set A of new type variables we introduce. These two relations, $e\colon t \Rightarrow_A C$ and $t /\!/\!/ p \Rightarrow_A (\Gamma, C)$, are defined by the rules in Figure A.2 and Figure A.3, respectively.

We use the symbol $\uplus$ to denote the union of two disjoint sets. Therefore when we write $A_1 \uplus A_2$ we require A_1 and A_2 to be disjoint. Since there is an infinite supply of type variables to choose from, the condition is always satisfiable by an appropriate choice of variables.

Lemma A.32: Variables in constraints *Given a constraint set C, we write $\mathit{var}(C)$ for the set of variables appearing in it. The following properties hold:*

- *whenever $t /\!/\!/ p \Rightarrow_A (\Gamma, C)$, we have $\mathit{var}(C) \subseteq \mathit{var}(t) \cup A$, $\mathit{var}(\Gamma) \subseteq \mathit{var}(t) \cup A$, and $A \sharp t$;*
- *whenever $e\colon t \Rightarrow_A C$, we have $\mathit{var}(C) \subseteq \mathit{var}(t) \cup A$ and $A \sharp t$;*
- *whenever $\Gamma \vdash C \rightsquigarrow D$, we have $\mathit{var}(D) \subseteq \mathit{var}(C) \cup \mathit{var}(\Gamma)$.*

$$\textit{TRs-Var}\ \frac{}{x\colon t \Rightarrow_{\varnothing} \{x \mathrel{\dot{\leq}} t\}} \qquad \textit{TRs-Const}\ \frac{}{c\colon t \Rightarrow_{\varnothing} \{c \mathrel{\dot{\leq}} t\}}$$

$$\textit{TRs-Abstr}\ \frac{e\colon \beta \Rightarrow_A C}{\lambda x.e\colon t \Rightarrow_{A \uplus \{\alpha,\beta\}} \{\mathsf{def}\ \{x\colon \alpha\}\ \mathsf{in}\ C, \alpha \to \beta \mathrel{\dot{\leq}} t\}}\ A, \alpha, \beta \nVdash t$$

$$\textit{TRs-Appl}\ \frac{e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha \Rightarrow_{A_2} C_2}{e_1\, e_2\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha,\beta\}} C_1 \cup C_2 \cup \{\beta \mathrel{\dot{\leq}} t\}}\ A_1, A_2, \alpha, \beta \nVdash t$$

$$\textit{TRs-Pair}\ \frac{e_1\colon \alpha_1 \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha_2 \Rightarrow_{A_2} C_2}{(e_1, e_2)\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1,\alpha_2\}} C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \mathrel{\dot{\leq}} t\}}\ A_1, A_2, \alpha_1, \alpha_2 \nVdash t$$

$$\textit{TRs-Tag}\ \frac{e\colon \alpha \Rightarrow_A C}{\grave{}tag(e)\colon t \Rightarrow_{A \uplus \{\alpha\}} C \cup \{\grave{}tag(\alpha) \mathrel{\dot{\leq}} t\}}\ A, \alpha \nVdash t$$

$$\textit{TRs-MatchM}\ \frac{\begin{array}{c} e_0\colon \alpha \Rightarrow_{A_0} C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \forall\, i \in I \qquad t_i /\!/\!/ p_i \Rightarrow_{A_i} (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow_{A'_i} C'_i \\ C = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\, \mathsf{def}\ \Gamma_i\ \mathsf{in}\ C'_i \mid i \in I \,\} \cup \{\alpha \mathrel{\dot{\leq}} \bigvee_{i\in I} \lceil p_i \rfloor, \beta \mathrel{\dot{\leq}} t\} \\ A = A_0 \uplus (\biguplus_{i\in I} A_i) \uplus (\biguplus_{i\in I} A'_i) \uplus \{\alpha, \beta\} \end{array}}{\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow_A C}\ A \nVdash t$$

FIGURE A.2 Constraint generation rules (without let-polymorphism) with explicit variables.

$$\frac{}{t /\!/\!/ \_ \Rightarrow_{\varnothing} (\varnothing, \varnothing)} \qquad \frac{}{t /\!/\!/ x \Rightarrow_{\varnothing} (\{x\colon t\}, \varnothing)} \qquad \frac{}{t /\!/\!/ c \Rightarrow_{\varnothing} (\varnothing, \varnothing)}$$

$$\frac{\alpha_1 /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad \alpha_2 /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t /\!/\!/ (p_1, p_2) \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1,\alpha_2\}} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \mathrel{\dot{\leq}} \alpha_1 \times \alpha_2\})}\ A_1, A_2, \alpha_1, \alpha_2 \nVdash t$$

$$\frac{\alpha /\!/\!/ p \Rightarrow_A (\Gamma, C)}{t /\!/\!/ \grave{}tag(p) \Rightarrow_{A \uplus \{\alpha\}} (\Gamma, C \cup \{t \mathrel{\dot{\leq}} \grave{}tag(\alpha)\})}\ A, \alpha \nVdash t$$

$$\frac{t /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad t /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t /\!/\!/ p_1 \& p_2 \Rightarrow_{A_1 \uplus A_2} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2)}$$

$$\frac{(t \wedge \lceil p_1 \rfloor) /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad (t \setminus \lceil p_1 \rfloor) /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)}{t /\!/\!/ p_1 | p_2 \Rightarrow_{A_1 \uplus A_2} (\{\, x\colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathit{capt}(p_1) \,\}, C_1 \cup C_2)}$$

FIGURE A.3 Constraint generation for pattern environments with explicit variables.

Proof Straightforward proofs by induction on the derivations. □

Pattern environment reconstruction

LEMMA A.33: Correctness of environment reconstruction *Let p be a pattern and t, t' two types, with $t' \leq \lfloor p \rfloor$. Let $t /\!/\!/ p \Rightarrow (\Gamma, C)$. If θ is a type substitution such that $\theta \Vdash C$ and $t' \leq t\theta$, then, for all $x \in \mathit{capt}(p)$, $(t' /\!/ p)(x) \leq \Gamma(x)\theta$.*

Proof By structural induction on p.

Cases $p = \_$ and $p = c$ There is nothing to prove since $\mathit{capt}(p) = \varnothing$.

Case $p = x$ We have

$$t /\!/\!/ x \Rightarrow (\{x : t\}, \varnothing) \qquad (t' /\!/ x)(x) = t'$$

and must show $t' \leq t\theta$, which we know by hypothesis.

Case $p = (p_1, p_2)$ We have

$$t /\!/\!/ p \Rightarrow (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\}) \qquad \forall i.\ \alpha_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i)\ .$$

Each $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Let $x \in \mathit{capt}(p_i)$; then, we must show $(\pi_i(t') /\!/ p_i)(x) \leq \Gamma_i(x)\theta$. This follows from the induction hypothesis, since $t' \leq t\theta \leq \alpha_1\theta \times \alpha_2\theta$ implies $\pi_i(t') \leq \alpha_i\theta$ by Property 4.10.

Case $p = \text{`}tag(p_1)$ We have

$$t /\!/\!/ \text{`}tag(p) \Rightarrow (\Gamma, C \cup \{t \dot{\leq} \text{`}tag(\alpha)\}) \qquad \alpha /\!/\!/ p \Rightarrow (\Gamma, C)\ .$$

Analogous to the previous case. We can apply the induction hypothesis, because $t' \leq t\theta \leq \text{`}tag(\alpha)\theta$ implies $\pi_{\text{`}tag}(t') \leq \alpha\theta$ by Property 4.11.

Case $p = p_1 \& p_2$ Every $x \in \mathit{capt}(p)$ is either in $\mathit{capt}(p_1)$ or in $\mathit{capt}(p_2)$. Let $x \in \mathit{capt}(p_i)$; then, we apply the induction hypothesis to p_i to conclude.

Case $p = p_1 | p_2$ We have

$$t /\!/\!/ p_1 | p_2 \Rightarrow (\{\, x : \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathit{capt}(p_1) \,\}, C_1 \cup C_2)$$
$$(t \wedge \lfloor p_1 \rfloor) /\!/\!/ p_1 \Rightarrow (\Gamma_1, C_1) \qquad (t \setminus \lfloor p_1 \rfloor) /\!/\!/ p_2 \Rightarrow (\Gamma_2, C_2)\ .$$

By the induction hypothesis applied to both p_1 and p_2 we derive, for all x,

$$(t' \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \leq \Gamma_1(x)\theta \qquad (t' \setminus \lfloor p_1 \rfloor /\!/ p_2)(x) \leq \Gamma_2(x)\theta$$

from which we can conclude

$$(t' /\!/ p)(x) = (t' \wedge \lfloor p_1 \rfloor /\!/ p_1)(x) \vee (t' \setminus \lfloor p_1 \rfloor /\!/ p_2)(x) \leq \Gamma_1(x)\theta \vee \Gamma_2(x)\theta\ .$$ □

LEMMA A.34: Precise solution to environment reconstruction constraints *Let p be a pattern, t a type, and θ a type substitution such that $t\theta \leq \lfloor p \rfloor$. Let $t /\!/\!/ p \Rightarrow_A (\Gamma, C)$, with $A \sharp \mathit{dom}(\theta)$.*

There exists a type substitution θ' such that $\mathit{dom}(\theta') = A$, that $(\theta \cup \theta') \Vdash C$, and that, for all $x \in \mathit{capt}(p)$, $\Gamma(x)(\theta \cup \theta') \leq (t\theta /\!/ p)(x)$.

Proof By structural induction on p.

Cases $p = \_$ and $p = c$ In both cases we take $\theta' = [\,]$.

Case $p = x$ We have

$$t /\!/\!/ x \Rightarrow_{\varnothing} (\{x\colon t\}, \varnothing)\,.$$

We take $\theta' = [\,]$ and have $t(\theta \cup \theta') \leq t\theta$.

Case $p = (p_1, p_2)$ We have

$$t /\!/\!/ (p_1, p_2) \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1, \alpha_2\}} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\})$$
$$\alpha_1 /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad \alpha_2 /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2) \qquad A_1, A_2, \alpha_1, \alpha_2 \sharp t\,.$$

Let $\theta^* = \theta \cup [{}^{\pi_1(t\theta)}/_{\alpha_1}, {}^{\pi_2(t\theta)}/_{\alpha_2}]$. We have $t\theta' = t\theta$ and $t\theta' \leq \lceil (p_1, p_2) \rfloor = \lceil p_1 \rfloor \times \lceil p_2 \rfloor$; thus, by Property 4.10, $\pi_i(t\theta') \leq \lceil p_i \rfloor$. We also have $A_i \sharp \mathit{dom}(\theta^*), \alpha_i$ for both i, since $\{\alpha_1, \alpha_2\}$ is disjoint from each A_i.

We can therefore apply the induction hypothesis to p_i, α_i, and θ^*, for both i. We derive from each that there is a substitution θ_i' with domain A_i, such that $(\theta^* \cup \theta_i') \Vdash C_i$ and, for all $x \in \mathit{capt}(p_i)$, $\Gamma_i(x)(\theta^* \cup \theta_i') \leq (\alpha_i\theta^* /\!/ p_i)(x)$.

We take $\theta' = [{}^{\pi_1(t\theta)}/_{\alpha_1}, {}^{\pi_2(t\theta)}/_{\alpha_2}] \cup \theta_1' \cup \theta_2'$. We have $(\theta \cup \theta') \Vdash C_1 \cup C_2 \cup \{t \dot{\leq} \alpha_1 \times \alpha_2\}$ since it satisfies C_1 and C_2 and since $t\theta \leq (\alpha_1 \times \alpha_2)\theta' = \pi_1(t\theta) \times \pi_2(t\theta)$.

Case $p = \grave{}\mathit{tag}(p_1)$ We have

$$t /\!/\!/ \grave{}\mathit{tag}(p_1) \Rightarrow_{A_1 \uplus \{\alpha\}} (\Gamma_1, C_1 \cup \{t \dot{\leq} \grave{}\mathit{tag}(\alpha)\}) \qquad \alpha /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad A_1, \alpha \sharp t\,.$$

Analogously to the previous case, we construct $\theta^* = \theta \cup [{}^{\pi_{\grave{}\mathit{tag}}(t\theta)}/_{\alpha}]$ and apply the induction hypothesis to p_1, α, and θ^*. We derive θ_1' and take $\theta' = [{}^{\pi_{\grave{}\mathit{tag}}(t\theta)}/_{\alpha}] \cup \theta_1'$.

Case $p = p_1 \& p_2$ We have

$$t /\!/\!/ p_1 \& p_2 \Rightarrow_{A_1 \uplus A_2} (\Gamma_1 \cup \Gamma_2, C_1 \cup C_2) \qquad t /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad t /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)\,.$$

For both i, we apply the induction hypothesis to p_i, t, and θ to derive θ_i'. We take $\theta' = \theta_1' \cup \theta_2'$.

Case $p = p_1 | p_2$ We have

$$t /\!/\!/ p_1 | p_2 \Rightarrow_{A_1 \uplus A_2} (\{\, x\colon \Gamma_1(x) \vee \Gamma_2(x) \mid x \in \mathit{capt}(x) \,\}, C_1 \cup C_2)$$
$$(t \wedge \lceil p_1 \rfloor) /\!/\!/ p_1 \Rightarrow_{A_1} (\Gamma_1, C_1) \qquad (t \setminus \lceil p_1 \rfloor) /\!/\!/ p_2 \Rightarrow_{A_2} (\Gamma_2, C_2)\,.$$

We apply the induction hypothesis to p_1, $t \wedge \lceil p_1 \rfloor$, and θ to derive θ_1'. We apply it to p_2, $t \setminus \lceil p_1 \rfloor$, and θ to derive θ_2'; here, note that $t\theta \leq \lceil p_1 \rfloor \vee \lceil p_2 \rfloor$ implies $t\theta \setminus \lceil p_1 \rfloor \leq \lceil p_2 \rfloor$.

We take $\theta' = \theta_1' \cup \theta_2'$. We have $(\theta \cup \theta') \Vdash C$ since it satisfies C_1 and C_2. Furthermore, for all x, we have $\Gamma_1(x)(\theta \cup \theta_1') \leq (t\theta \wedge \lceil p_1 \rfloor /\!/ p_1)(x)$ and $\Gamma_2(x)(\theta \cup \theta_2') \leq (t\theta \setminus \lceil p_1 \rfloor /\!/ p_2)(x)$. Then, $\Gamma(x)(\theta \cup \theta') = \Gamma_1(x)(\theta \cup \theta') \vee \Gamma_2(x)(\theta \cup \theta') = \Gamma_1(x)(\theta \cup \theta_1') \vee \Gamma_2(x)(\theta \cup \theta_2')$, since A_1 and A_2 are disjoint and both are disjoint from $\mathit{var}(t)$. Finally, $\Gamma_1(x)(\theta \cup \theta_1') \vee \Gamma_2(x)(\theta \cup \theta_2') \leq (t\theta /\!/ p)(x)$. □

Reconstruction without let-polymorphism

THEOREM A.35: Soundness of constraint generation and rewriting *Let e be an expression, t a type, and Γ a type environment. If $e\colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_{\mathrm{s}} e\colon t\theta$.*

Proof By structural induction on e.

Case $e = x$ We have

$$x\colon t \Rightarrow \{x \dot{\leq} t\} \qquad \Gamma \vdash \{x \dot{\leq} t\} \rightsquigarrow \{t_x \dot{\leq} t\} \qquad \Gamma(x) = t_x\,.$$

By *Ts-Var* we derive $\Gamma\theta \vdash_{\mathrm{s}} x\colon t_x\theta$. Since $\theta \Vdash D$, and hence $t_x\theta \leq t\theta$, we have $\Gamma\theta \vdash_{\mathrm{s}} x\colon t\theta$ by subsumption.

Case $e = c$ We have

$$c\colon t \Rightarrow \{c \dot{\leq} t\} \qquad \Gamma \vdash \{c \dot{\leq} t\} \rightsquigarrow \{c \dot{\leq} t\}\,.$$

Analogously to the previous case, we first apply *Ts-Const* and then conclude by subsumption.

Case $e = \lambda x.e_1$ We have

$$\lambda x.e_1\colon t \Rightarrow \{\mathsf{def}\,\{x\colon \alpha\}\,\mathsf{in}\,C_1, \alpha \to \beta \dot{\leq} t\} \qquad e_1\colon \beta \Rightarrow C_1$$
$$\Gamma \vdash \{\mathsf{def}\,\{x\colon \alpha\}\,\mathsf{in}\,C_1, \alpha \to \beta \dot{\leq} t\} \rightsquigarrow D_1 \cup \{\alpha \to \beta \dot{\leq} t\} \qquad \Gamma, \{x\colon \alpha\} \vdash C_1 \rightsquigarrow D_1\,.$$

By the induction hypothesis we derive $\Gamma\theta, \{x\colon \alpha\theta\} \vdash_{\mathrm{s}} e_1\colon \beta\theta$. We apply *Ts-Abstr* to derive $\Gamma\theta \vdash_{\mathrm{s}} \lambda x.e_1\colon (\alpha \to \beta)\theta$. Since $\theta \Vdash D$, we have $(\alpha \to \beta)\theta \leq t\theta$. Hence, we derive by subsumption $\Gamma\theta \vdash_{\mathrm{s}} \lambda x.e_1\colon t\theta$.

Case $e = e_1\, e_2$ We have

$$e_1\, e_2\colon t \Rightarrow C_1 \cup C_2 \cup \{\beta \dot{\leq} t\} \qquad e_1\colon \alpha \to \beta \Rightarrow C_1 \qquad e_2\colon \alpha \Rightarrow C_2$$
$$\Gamma \vdash C_1 \cup C_2 \cup \{\beta \dot{\leq} t\} \rightsquigarrow D_1 \cup D_2 \cup \{\beta \dot{\leq} t\} \qquad \Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2\,.$$

We derive $\Gamma\theta \vdash_{\mathrm{s}} e_1\colon (\alpha\theta) \to (\beta\theta)$ and $\Gamma\theta \vdash_{\mathrm{s}} e_2\colon \alpha\theta$ by the induction hypothesis. Then by *Ts-Appl* we derive $\Gamma\theta \vdash_{\mathrm{s}} e_1\, e_2\colon \beta\theta$, and finally – since $\beta\theta \leq t\theta$ – we conclude by subsumption.

Case $e = (e_1, e_2)$ We have

$$(e_1, e_2)\colon t \Rightarrow C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\} \qquad e_1\colon \alpha_1 \Rightarrow C_1 \qquad e_2\colon \alpha_2 \Rightarrow C_2$$
$$\Gamma \vdash C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\} \rightsquigarrow D_1 \cup D_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\}$$
$$\Gamma \vdash C_1 \rightsquigarrow D_1 \qquad \Gamma \vdash C_2 \rightsquigarrow D_2\,.$$

We have $\Gamma\theta \vdash_{\mathrm{s}} e_i\colon \alpha_i\theta$ for both i by the induction hypothesis. Then, we derive $\Gamma\theta \vdash_{\mathrm{s}} (e_1, e_2)\colon (\alpha_1 \times \alpha_2)\theta$ by *Ts-Pair*, and finally conclude by subsumption.

Case $e = \text{`}tag(e_1)$ We have

$$\text{`}tag(e_1)\colon t \Rightarrow C \qquad C = C_1 \cup \{\text{`}tag(\alpha) \dot{\leq} t\} \qquad e_1\colon \alpha \Rightarrow C_1$$
$$\Gamma \vdash C \rightsquigarrow D \qquad D = D_1 \cup \{\text{`}tag(\alpha) \dot{\leq} t\} \qquad \Gamma \vdash C_1 \rightsquigarrow D_1\,.$$

Analogous to the previous case. We apply the induction hypothesis, then *Ts-Tag*, then subsumption.

Case $e = \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}$ We have

$$\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow C$$
$$C = C_0 \cup (\bigcup_{i\in I} C_i) \cup \{\, \mathsf{def}\ \Gamma_i\ \mathsf{in}\ C'_i \mid i \in I \,\} \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lceil p_i \rfloor, \beta \dot{\leq} t\} \qquad e_0\colon \alpha \Rightarrow C_0$$
$$\forall i \in I \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow C'_i$$
$$\Gamma \vdash C \rightsquigarrow D$$
$$D = D_0 \cup (\bigcup_{i\in I} D_i) \cup (\bigcup_{i\in I} D'_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lceil p_i \rfloor, \beta \dot{\leq} t\} \qquad \Gamma \vdash C_0 \rightsquigarrow D_0$$
$$\forall i \in I \qquad \Gamma \vdash C_i \rightsquigarrow D_i \qquad \Gamma, \Gamma_i \vdash C'_i \rightsquigarrow D'_i\,.$$

By the induction hypothesis we have $\Gamma\theta \vdash_{\mathrm{s}} e_0\colon \alpha\theta$ and $\Gamma\theta, \Gamma_i\theta \vdash_{\mathrm{s}} e_i\colon \beta\theta$ for all i. Exhaustiveness is satisfied since $\theta \Vdash \alpha \dot{\leq} \bigvee_{i\in I} \lceil p_i \rfloor$ and therefore $\alpha\theta \leq \bigvee_{i\in I} \lceil p_i \rfloor$ (all pattern types are closed).

We prove $\Gamma\theta, (t_i\theta)/\!/p_i \vdash_{\mathrm{s}} e_i\colon \beta\theta$ from $\Gamma\theta, \Gamma_i\theta \vdash_{\mathrm{s}} e_i\colon \beta\theta$, for all i, by Lemma A.20. To apply the lemma, we must show $\Gamma\theta, (t_i\theta)/\!/p_i \sqsubseteq \Gamma\theta, \Gamma_i\theta$, which holds because, by Lemma A.33, $((t_i\theta)/\!/p_i)(x) \leq (\Gamma_i\theta)(x)$ for any x. The second hypothesis of weakening (containment of variables) is not necessary in the absence of let-polymorphism, as it only influences the proof when generalization is concerned.

We can therefore apply *Ts-Match* to derive $\Gamma\theta \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \beta\theta$ and conclude by subsumption. □

THEOREM A.36: Completeness of constraint generation and rewriting *Let e be an expression, t a type, and Γ a type environment. Let θ be a type substitution such that $\Gamma\theta \vdash_{\mathrm{s}} e\colon t\theta$.*

Let $e\colon t \Rightarrow_A C$, with $A \sharp \Gamma, \mathit{dom}(\theta)$. There exist a type-constraint set D and a type substitution θ', with $\mathit{dom}(\theta') = A$, such that $\Gamma \vdash C \rightsquigarrow D$ and $(\theta \cup \theta') \Vdash D$.

Proof By structural induction on e.

Case $e = x$ We have

$$x\colon t \Rightarrow_\varnothing \{x \dot{\leq} t\} \qquad \Gamma\theta \vdash_{\mathrm{s}} x\colon t\theta \qquad \Gamma(x) = t_x \qquad t_x\theta \leq t\theta\,.$$

Since $x \in \mathit{dom}(\Gamma)$, we have $\Gamma \vdash \{x \dot{\leq} t\} \rightsquigarrow \{t_x \dot{\leq} t\}$. Let $\theta' = [\,]$; we have $(\theta \cup \theta') \Vdash \{t_x \dot{\leq} t\}$.

Case $e = c$ We have

$$c\colon t \Rightarrow_\varnothing \{c \dot{\leq} t\} \qquad \Gamma\theta \vdash_{\mathrm{s}} c\colon t\theta \qquad c \leq t\theta\,.$$

We have $\Gamma \vdash \{c \dot{\leq} t\} \rightsquigarrow \{c \dot{\leq} t\}$. Let $\theta' = [\,]$. We have $(\theta \cup \theta') \Vdash \{c \dot{\leq} t\}$ because $c\theta = c \leq t\theta$.

Case $e = \lambda x.e_1$ We have

$$\lambda x.e_1\colon t \Rightarrow_{A_1 \uplus \{\alpha,\beta\}} \{\mathsf{def}\ \{x\colon \alpha\}\ \mathsf{in}\ C_1, \alpha \to \beta \dot{\leq} t\} \qquad e_1\colon \beta \Rightarrow_{A_1} C_1 \qquad A_1, \alpha, \beta \sharp t$$
$$\Gamma\theta \vdash_{\mathrm{s}} \lambda x.e_1\colon t\theta \qquad \Gamma\theta, \{x\colon t_1\} \vdash_{\mathrm{s}} e_1\colon t_2 \qquad t_1 \to t_2 \leq t\theta\,.$$

Let $\theta^* = \theta \cup [t_1/\alpha, t_2/\beta]$. Note that $\Gamma\theta^* = \Gamma\theta$ and $t\theta^* = t\theta$, because $\{\alpha_1, \alpha_2\} \sharp \Gamma, t$.

We have $(\Gamma, \{x\colon \alpha\})\theta^* \vdash_{\mathrm{s}} e_1\colon \beta\theta^*$, $e_1\colon \beta \Rightarrow_{A_1} C_1$, and $A_1 \sharp \mathit{dom}(\theta^*)$. By the induction hypothesis, therefore, $\Gamma, \{x\colon \alpha\} \vdash C_1 \rightsquigarrow D_1$ and $(\theta^* \cup \theta'_1) \Vdash D_1$, for some D_1 and θ'_1 with $\mathit{dom}(\theta'_1) = A_1$.

$\Gamma, \{x\colon \alpha\} \vdash C_1 \rightsquigarrow D_1$ implies $\Gamma \vdash \mathsf{def}\ \{x\colon \alpha\}\ \mathsf{in}\ C_1 \rightsquigarrow D_1$. Hence, we have $\Gamma \vdash C \rightsquigarrow D = D_1 \cup \{\alpha \to \beta \dot{\leq} t\}$. Let $\theta' = [t_1/\alpha, t_2/\beta] \cup \theta'_1$. It is defined on the correct domain and it solves the constraints, since it solves D_1 and since $(\alpha \to \beta)\theta' = t_1 \to t_2 \leq t\theta$.

Case $e = e_1\ e_2$ We have

$$\begin{array}{l} e_1\ e_2\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha,\beta\}} C_1 \cup C_2 \cup \{\beta \dot{\leq} t\} \\ \quad e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha \Rightarrow_{A_2} C_2 \qquad A_1, A_2, \alpha, \beta \sharp t \\ \Gamma\theta \vdash_{\mathrm{s}} e_1\ e_2\colon t\theta \qquad \Gamma\theta \vdash_{\mathrm{s}} e_1\colon t_1 \to t_2 \qquad \Gamma\theta \vdash_{\mathrm{s}} e_2\colon t_1 \qquad t_2 \leq t\theta\,. \end{array}$$

Let $\theta^* = \theta \cup [t_1/\alpha, t_2/\beta]$. Note that $\Gamma\theta^* = \Gamma\theta$ and $t\theta^* = t\theta$, since $\alpha, \beta \sharp \Gamma, t$.

We have $\Gamma\theta \vdash_{\mathrm{s}} e_1\colon (\alpha \to \beta)\theta^*$, $e_1\colon \alpha \to \beta \Rightarrow_{A_1} C_1$, and $A_1 \sharp \mathit{dom}(\theta^*)$. By the induction hypothesis, therefore, $\Gamma \vdash C_1 \rightsquigarrow D_1$ and $(\theta^* \cup \theta_1') \Vdash D_1$, for some D_1 and θ_1' with $\mathit{dom}(\theta_1') = A_1$.

Likewise, by applying the induction hypothesis to the derivation for e_2, we derive $\Gamma \vdash C_2 \rightsquigarrow D_2$ and $(\theta^* \cup \theta_2') \Vdash D_2$, for some D_2 and θ_2' with $\mathit{dom}(\theta_2') = A_2$.

We can thus conclude that $\Gamma \vdash C \rightsquigarrow D = D_1 \cup D_2 \cup \{\beta \dot{\leq} t\}$. Let $\theta' = [t_1/\alpha, t_2/\beta] \cup \theta_1' \cup \theta_2'$. It is defined on the correct domain and $\theta \cup \theta'$ solves the constraints: it solves both D_1 and D_2, and $\beta(\theta \cup \theta') = \beta\theta' = t_2 \leq t\theta = t(\theta \cup \theta')$.

Case $e = (e_1, e_2)$ We have

$$\begin{array}{l} (e_1, e_2)\colon t \Rightarrow_{A_1 \uplus A_2 \uplus \{\alpha_1,\alpha_2\}} C_1 \cup C_2 \cup \{\alpha_1 \times \alpha_2 \dot{\leq} t\} \\ \quad e_1\colon \alpha_1 \Rightarrow_{A_1} C_1 \qquad e_2\colon \alpha_2 \Rightarrow_{A_2} C_2 \qquad A_1, A_2, \alpha_1, \alpha_2 \sharp t \\ \Gamma\theta \vdash_{\mathrm{s}} (e_1, e_2)\colon t\theta \qquad \Gamma\theta \vdash_{\mathrm{s}} e_1\colon t_1 \qquad \Gamma\theta \vdash_{\mathrm{s}} e_2\colon t_2 \qquad t_1 \times t_2 \leq t\theta\,. \end{array}$$

Analogous to the previous case. We define $\theta^* = \theta \cup [t_1/\alpha_1, t_2/\alpha_2]$ and proceed as above.

Case $e = \grave{}\mathit{tag}(e_1)$ We have

$$\begin{array}{l} \grave{}\mathit{tag}(e_1)\colon t \Rightarrow_{A_1 \uplus \{\alpha\}} C_1 \cup \{\grave{}\mathit{tag}(\alpha) \dot{\leq} t\} \qquad e_1\colon \alpha \Rightarrow_{A_1} C_1 \qquad A_1, \alpha \sharp t \\ \Gamma\theta \vdash_{\mathrm{s}} \grave{}\mathit{tag}(e)\colon t\theta \qquad \Gamma\theta \vdash_{\mathrm{s}} e_1\colon t_1 \qquad \grave{}\mathit{tag}(t_1) \leq t\theta\,. \end{array}$$

Analogous to the two previous cases. Here we define $\theta^* = \theta \cup [t_1/\alpha]$.

Case $e = \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I}$ We have

$$\begin{array}{l} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I}\colon t \Rightarrow_A C \\ \quad C = C_0 \cup (\bigcup_{i \in I} C_i) \cup \{\, \mathsf{def}\ \Gamma_i\ \mathsf{in}\ C_i' \mid i \in I \,\} \cup \{\alpha \dot{\leq} \bigvee_{i \in I} \lceil p_i \rfloor, \beta \dot{\leq} t\} \\ \quad A = A_0 \uplus (\biguplus_{i \in I} A_i) \uplus (\biguplus_{i \in I} A_i') \uplus \{\alpha, \beta\} \\ \quad e_0\colon \alpha \Rightarrow_{A_0} C_0 \qquad t_i = (\alpha \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \quad \forall i \in I \qquad t_i /\!/\!/ p_i \Rightarrow_{A_i} (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow_{A_i'} C_i' \qquad A \sharp t \\ \Gamma\theta \vdash_{\mathrm{s}} \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i \in I}\colon t\theta \\ \quad \Gamma\theta \vdash_{\mathrm{s}} e_0\colon t_0 \qquad t_0 \leq \bigvee_{i \in I} \lceil p_i \rfloor \qquad t_i^* = (t_0 \setminus \bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \\ \quad \forall i \in I \qquad \Gamma\theta, t_i^* /\!/ p_i \vdash_{\mathrm{s}} e_i\colon t_i' \qquad t' = \bigvee_{i \in I} t_i' \leq t\theta\,. \end{array}$$

Let $\theta^* = \theta \cup [t_0/\alpha, t'/\beta]$. Note that $\Gamma\theta^* = \Gamma\theta$ and $t\theta^* = t\theta$. Also, $t_i\theta^* = t_i^*$ for all i.

By the induction hypothesis, there are D_0 and θ_0' such that $\Gamma \vdash C_0 \rightsquigarrow D_0$, $\mathit{dom}(\theta_0') = A_0$, and $(\theta^* \cup \theta_0') \Vdash D_0$. Note also that $\theta^* \Vdash \{\alpha \dot{\leq} \bigvee_{i \in I} \lceil p_i \rfloor\}$, because $\alpha\theta^* = t_0 \leq \bigvee_{i \in I} \lceil p_i \rfloor$.

For each i, by Lemma A.34 (again considering θ^*), there exists a θ_i' such that $\mathit{dom}(\theta_i') = A_i$, $(\theta^* \cup \theta_i') \Vdash C_i$, and, for all $x \in \mathit{capt}(p_i)$, $\Gamma_i(x)(\theta^* \cup \theta_i') \leq (t_i^* /\!/ p_i)(x)$. Note that $\Gamma(\theta^* \cup \theta_i') = \Gamma\theta$. By Lemma A.20, from $\Gamma\theta, t_i^* /\!/ p_i \vdash_{\mathrm{s}} e_i\colon t_i'$ we derive $(\Gamma, \Gamma_i)(\theta^* \cup \theta_i') \vdash_{\mathrm{s}} e_i\colon t_i'$; then, by subsumption, we derive $(\Gamma, \Gamma_i)(\theta^* \cup \theta_i') \vdash_{\mathrm{s}} e_i\colon t'$.

Now we have $e_i \colon \beta \Rightarrow_{A_i'} C_i'$ and $(\Gamma, \Gamma_i)(\theta^* \cup \theta_i') \vdash_{\mathrm{s}} e_i \colon \beta(\theta^* \cup \theta_i')$. By the induction hypothesis, there are D_i' and θ_i'' such that $\Gamma, \Gamma_i \vdash C_i' \rightsquigarrow D_i'$ and $(\theta^* \cup \theta_i' \cup \theta_i'') \Vdash C_i'$.

Since $\Gamma, \Gamma_i \vdash C_i' \rightsquigarrow D_i'$ implies $\Gamma \vdash \mathsf{def}\ \Gamma_i\ \mathsf{in}\ C_i' \rightsquigarrow D_i'$, we have $\Gamma \vdash C \rightsquigarrow D_0 \cup (\bigcup_{i \in I} D_i) \cup (\bigcup_{i \in I} D_i') \cup \{\alpha \mathrel{\dot{\leq}} \bigvee_{i \in I} \lceil p_i \rfloor, \beta \mathrel{\dot{\leq}} t\}$ and we take $\theta' = [{}^{t_0}\!/_{\alpha}, {}^{t'}\!/_{\beta}] \cup (\bigcup_{i \in I} \theta_i') \cup (\bigcup_{i \in I} \theta_i'')$. □

Adding let-polymorphism

We first prove a different version of the weakening lemma (Lemma A.20) which is more general than the original statement (its proof relies on Lemma A.21, whose proof in turn used Lemma A.20).

LEMMA A.37 *Let Γ_1, Γ_2 be two type environments such that $\Gamma_1 \sqsubseteq \Gamma_2$. Let t be a type such that $\mathit{var}(\Gamma_1) \cap \mathit{var}(t) \subseteq \mathit{var}(\Gamma_2)$. If $\Gamma_2 \vdash_{\mathrm{s}} e \colon t$, then $\Gamma_1 \vdash_{\mathrm{s}} e \colon t$.*

Proof By induction on the derivation of $\Gamma_2 \vdash_{\mathrm{s}} e \colon t$. We reason by cases on the last applied rule.

Cases Ts-Var and Ts-Const As in the proof of Lemma A.20.

Case Ts-Abstr We have

$$\Gamma_2 \vdash_{\mathrm{s}} \lambda x.e_1 \colon t_1 \to t_2 \qquad \Gamma_2, \{x \colon t_1\} \vdash_{\mathrm{s}} e_1 \colon t_2\ .$$

Since $\Gamma_1 \sqsubseteq \Gamma_2$, we have $\Gamma_1, \{x \colon t_1\} \sqsubseteq \Gamma_2, \{x \colon t_1\}$. Since $\mathit{var}(\Gamma_1) \cap \mathit{var}(t_1 \to t_2) \subseteq \mathit{var}(\Gamma_2)$, we have $\mathit{var}(\Gamma_1, \{x \colon t_1\}) \cap \mathit{var}(t_2) \subseteq \mathit{var}(\Gamma_2, \{x \colon t_1\})$. We derive $\Gamma_1, \{x \colon t_1\} \vdash_{\mathrm{s}} e_1 \colon t_2$ by the induction hypothesis and conclude by *Ts-Abstr*.

Case Ts-Appl We have

$$\Gamma_2 \vdash_{\mathrm{s}} e_1\ e_2 \colon t \qquad \Gamma_2 \vdash_{\mathrm{s}} e_1 \colon t' \to t \qquad \Gamma_2 \vdash_{\mathrm{s}} e_2 \colon t'\ .$$

Let θ be a type substitution which maps the variables in $\mathit{var}(t') \setminus (\mathit{var}(t) \cup \mathit{var}(\Gamma_2))$ into other variables which do not appear in Γ_1. Note that $t\theta = t$ and $\Gamma_2\theta = \Gamma_2$. By Lemma A.21 we have

$$\Gamma_2 \vdash_{\mathrm{s}} e_1 \colon (t'\theta) \to t \qquad \Gamma_2 \vdash_{\mathrm{s}} e_2 \colon t'\theta\ .$$

We have $\mathit{var}(\Gamma_1) \cap \mathit{var}(t'\theta) \subseteq \mathit{var}(\Gamma_1) \cap \mathit{var}((t'\theta) \to t) \subseteq \mathit{var}(\Gamma_2)$ and $\Gamma_1 \sqsubseteq \Gamma_2$. Hence, by the induction hypothesis, we derive $\Gamma_2 \vdash_{\mathrm{s}} e_1 \colon (t'\theta) \to t$ and $\Gamma_2 \vdash_{\mathrm{s}} e_2 \colon t'\theta$. We apply *Ts-Appl* to conclude.

Cases Ts-Pair and Ts-Tag Straightforward application of the induction hypothesis.

Cases Ts-Match and Ts-Subsum Analogous to the case for applications. We replace the variables in t_0 or t', respectively, with new variables if they do not appear also in t or in Γ_2. □

THEOREM A.38: Soundness of constraint generation and rewriting with let-polymorphism
Let e be an expression, t a type, and Γ a type environment. If $e \colon t \Rightarrow C$, $\Gamma \vdash C \rightsquigarrow D$, and $\theta \Vdash D$, then $\Gamma\theta \vdash_{\mathrm{s}} e \colon t\theta$.

Proof By structural induction on e. We write only the cases for variables and pattern-matching expressions; the others are as in the proof of Theorem A.35.

Case $e = x$ We have

$$x\colon t \Rightarrow \{x \dot{\leq} t\}$$
$$\Gamma \vdash \{x \dot{\leq} t\} \rightsquigarrow \{t_x[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \dot{\leq} t\} \qquad \Gamma(x) = \forall\{\alpha_1, \ldots, \alpha_n\}.\, t_x\ .$$

Let $A = \{\alpha_1, \ldots, \alpha_n\}$. By α-renaming we assume $A \sharp \theta$; then we have $(\Gamma\theta)(x) = (\forall A.\, t_x)\theta = \forall A.\, (t_x\theta)$. Consider the substitution $\theta_x = [\beta_1\theta/\alpha_1, \ldots, \beta_n\theta/\alpha_n]$. It has domain A, so we can derive $\Gamma\theta \vdash_s x\colon t_x\theta\theta_x$.

We show $t_x\theta\theta_x = t_x[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\theta$ by showing $\alpha\theta\theta_x = \alpha[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\theta$ holds for all $\alpha \in \mathit{var}(t_x)$. Either $\alpha \in A$ or not. In the first case, $\alpha = \alpha_i$ for some i; then $\alpha\theta\theta_x = \alpha\theta_x = \beta_i\theta$ and $\alpha[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\theta = \beta_i\theta$. In the latter, $\alpha \neq \alpha_i$ for all i; then $\alpha\theta\theta_x = \alpha\theta$, since $\mathit{var}(\alpha\theta) \cap \mathit{dom}(\theta_x) = \varnothing$ and $\alpha[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\theta = \alpha\theta$.

Therefore we derive $\Gamma\theta \vdash_s x\colon t_x[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]\theta$ by *Ts-Var* and $\Gamma\theta \vdash_s x\colon t\theta$ by subsumption.

Case $e = \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}$ We have

$$\mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t \Rightarrow C$$
$$C = \{\mathsf{let}\ [C_0'](\Gamma_i\ \mathsf{in}\ C_i')_{i\in I}, \beta \dot{\leq} t\}$$
$$C_0' = C_0 \cup (\textstyle\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lceil p_i \rfloor\} \qquad e_0\colon \alpha \Rightarrow C_0$$
$$\forall i \in I \qquad t_i = (\alpha \setminus \textstyle\bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor \qquad t_i /\!/\!/ p_i \Rightarrow (\Gamma_i, C_i) \qquad e_i\colon \beta \Rightarrow C_i'$$
$$\Gamma \vdash C \rightsquigarrow D$$
$$\Gamma \vdash C_0' \rightsquigarrow D_0' \qquad \Gamma \vdash C_0 \rightsquigarrow D_0 \qquad D_0' = D_0 \cup (\textstyle\bigcup_{i\in I} C_i) \cup \{\alpha \dot{\leq} \bigvee_{i\in I} \lceil p_i \rfloor\}$$
$$\theta_0 \in \mathit{tally}(D_0') \qquad \forall i \in I.\ \ \Gamma, \mathit{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i' \rightsquigarrow D_i'$$
$$D = \mathit{equiv}(\theta_0) \cup (\textstyle\bigcup_{i\in I} D_i') \cup \{\beta \dot{\leq} t\}$$

and we must show $\Gamma\theta \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t\theta$.

We prove it by establishing, for some types $\hat{t}_0$ and $\hat{t}_i$, $\hat{t}_i'$ for each i, that

$$\Gamma\theta \vdash_s e_0\colon \hat{t}_0 \qquad \hat{t}_0 \leq \textstyle\bigvee_{i\in I} \lceil p_i \rfloor \qquad \hat{t}_i = (\hat{t}_0 \setminus \textstyle\bigvee_{j<i} \lceil p_j \rfloor) \wedge \lceil p_i \rfloor$$
$$\forall i \in I.\ \ \Gamma\theta, \mathit{gen}_{\Gamma\theta}(\hat{t}_i /\!/ p_i) \vdash_s e_i\colon \hat{t}_i' \qquad \textstyle\bigvee_{\hat{t}_i \not\leq \mathbb{0}} \hat{t}_i' \leq t\theta\ .$$

Since $\theta_0 \in \mathit{tally}(D_0')$, $\theta_0 \Vdash D_0'$ and thus $\theta_0 \Vdash D_0$. Then, from

$$e_0\colon \alpha \Rightarrow C_0 \qquad \Gamma \vdash C_0 \rightsquigarrow D_0 \qquad \theta_0 \Vdash D_0$$

we derive $\Gamma\theta_0 \vdash_s e_0\colon \alpha\theta_0$ by the induction hypothesis.

Let $A = \mathit{var}(\alpha\theta_0) \setminus \mathit{var}(\Gamma\theta_0) = \{\alpha_1, \ldots, \alpha_n\}$. Let $B = \{\beta_1, \ldots, \beta_n\}$ be a set of type variables such that $B \sharp \Gamma, \theta, \theta_0$ and let $\theta^* = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$. We derive $\Gamma\theta_0\theta^* \vdash_s e_0\colon \alpha\theta_0\theta^*$ by Lemma A.21; note that $\Gamma\theta_0\theta^* = \Gamma\theta_0$, because we are substituting variables that do not appear in $\Gamma\theta_0$. By the same lemma, we derive $\Gamma\theta_0\theta \vdash_s e_0\colon \alpha\theta_0\theta^*\theta$. By Lemma A.37, we derive $\Gamma\theta \vdash_s e_0\colon \alpha\theta_0\theta^*\theta$ (we prove the required premises below).

We take $\hat{t}_0 = \alpha\theta_0\theta^*\theta$. We have $\alpha\theta_0\theta^*\theta \leq \bigvee_{i\in I} \lceil p_i \rfloor$ because $\theta_0 \Vdash D_0'$ implies $\alpha\theta_0 \leq \bigvee_{i\in I} \lceil p_i \rfloor$ and because subtyping is preserved by substitutions (recall that the accepted types of patterns are closed). We also have $\hat{t}_i = t_i\theta_0\theta^*\theta$ for all i.

For each branch i, from

$$e_i\colon \beta \Rightarrow C_i' \qquad \Gamma, \mathit{gen}_{\Gamma\theta_0}(\Gamma_i\theta_0) \vdash C_i' \rightsquigarrow D_i' \qquad \theta \Vdash D_i'$$

we derive $\Gamma\theta, (gen_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta \vdash_s e_i\colon \beta\theta$ by the induction hypothesis. We derive by Lemma A.20 $\Gamma\theta, gen_{\Gamma\theta}(\hat{t}_i/\!/p_i) \vdash_s e_i\colon \beta\theta$ (we prove the premises below). Thus we have $\hat{t}'_i = \beta\theta$ for every branch; we apply *Ts-Match′* to derive $\Gamma\theta \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon \beta\theta$, then subsumption to derive $\Gamma\theta \vdash_s \mathsf{match}\ e_0\ \mathsf{with}\ (p_i \to e_i)_{i\in I}\colon t\theta$.

Proof of $\Gamma\theta \vdash_s e_0\colon \alpha\theta_0\theta^*\theta$ *from* $\Gamma\theta_0\theta \vdash_s e_0\colon \alpha\theta_0\theta^*\theta$ To apply Lemma A.37, we must show

$$\Gamma\theta \sqsubseteq \Gamma\theta_0\theta \qquad var(\Gamma\theta) \cap var(\alpha\theta_0\theta^*\theta) \subseteq var(\Gamma\theta_0\theta)\ .$$

To prove $\Gamma\theta \sqsubseteq \Gamma\theta_0\theta$, consider an arbitrary $x\colon \forall A_x.\, t_x \in \Gamma$. By α-renaming, we assume $A_x \sharp \theta, \theta_0$; then, we must prove $\forall A_x.\, t_x\theta \sqsubseteq \forall A_x.\, t_x\theta_0\theta$. For every γ, $\gamma\theta \simeq \gamma\theta_0\theta$ since $\theta \Vdash equiv(\theta_0)$. Hence, $t_x\theta \simeq t_x\theta_0\theta$.

To prove $var(\Gamma\theta) \cap var(\alpha\theta_0\theta^*\theta) \subseteq var(\Gamma\theta_0\theta)$, consider $\gamma \in var(\Gamma\theta) \cap var(\alpha\theta_0\theta^*\theta)$. Since $\gamma \in var(\alpha\theta_0\theta^*\theta)$, we have $\gamma \in var(\delta\theta)$ for some $\delta \in var(\alpha\theta_0\theta^*)$. We have $\delta \notin B$: if δ were in B, then it would be unaffected by θ, we would have $\gamma = \delta$, and $\gamma \in var(\Gamma\theta)$ could not hold. Since we have a $\delta \in var(\alpha\theta_0\theta^*)$, necessarily there must be a $\zeta \in var(\alpha\theta_0)$ such that $\delta \in var(\zeta\theta^*)$ and, since $\delta \notin B$, $\zeta \notin A$. Then $\zeta = \delta$, since θ^* only acts on variables in A. Thus we know that there exists a $\delta \in var(\alpha\theta_0)$ such that $\delta \notin A$: hence, $\delta \in var(\Gamma\theta_0)$. Then, since $\gamma \in var(\delta\theta)$, $\gamma \in var(\Gamma\theta_0\theta)$.

Proof of $\Gamma\theta, gen_{\Gamma\theta}(\hat{t}_i/\!/p_i) \vdash_s e_i\colon \beta\theta$ *from* $\Gamma\theta, (gen_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta \vdash_s e_i\colon \beta\theta$ By Lemma A.20, we can prove the result by showing

$$\Gamma\theta, gen_{\Gamma\theta}(\hat{t}_i/\!/p_i) \sqsubseteq \Gamma\theta, (gen_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta$$
$$var(\Gamma\theta, gen_{\Gamma\theta}(\hat{t}_i/\!/p_i)) \subseteq var(\Gamma\theta, (gen_{\Gamma\theta_0}(\Gamma_i\theta_0))\theta)\ .$$

The second condition is straightforward. For the first, we prove, for every $x \in capt(p_i)$, $gen_{\Gamma\theta}((\hat{t}_i/\!/p_i)(x)) \sqsubseteq (gen_{\Gamma\theta_0}(\Gamma_i\theta_0(x)))\theta$. Let $\Gamma_i(x) = t_x$. Then, $gen_{\Gamma\theta_0}(\Gamma_i\theta_0(x)) = \forall A.\, t_x\theta_0$, where A is $var(\alpha\theta_0) \setminus var(\Gamma\theta_0)$ as defined above (not all variables in A appear in $t_x\theta_0$, schemes are defined disregarding useless quantification). By α-renaming, we have $gen_{\Gamma\theta_0}(\Gamma_i\theta_0(x)) = \forall B.\, t_x\theta_0\theta^*$ and, since $B \sharp \theta$, $(gen_{\Gamma\theta_0}(\Gamma_i\theta_0(x)))\theta = \forall B.\, t_x\theta_0\theta^*\theta$.

Since $\hat{t}_i \le \hat{t}_0 = \alpha\theta_0\theta^*\theta$ and since $\theta \circ \theta^* \circ \theta_0 \Vdash C_i$ (because $\theta_0 \Vdash C_i$), by Lemma A.33 we have $(\hat{t}_i/\!/p_i)(x) \le t_x\theta_0\theta^*\theta$. Then, $gen_{\Gamma\theta}((\hat{t}_i/\!/p_i)(x)) \sqsubseteq \forall B.\, t_x\theta_0\theta^*\theta$ holds because all variables in B may be quantified when generalizing $(\hat{t}_i/\!/p_i)(x)$, since no β_i appears in $\Gamma\theta$. □

References

Balat, Vincent, Jérôme Vouillon, and Boris Yakobowski. Experience report: Ocsigen, a web programming framework. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Edinburgh, UK*, pages 311–316. 2009.

Benzaken, Véronique, Giuseppe Castagna, Kim Nguyễn, and Jérôme Siméon. Static and dynamic semantics of NoSQL languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 101–114. January 2013.

Benzaken, Véronique, Giuseppe Castagna, and Alain Frisch. ℂDuce: an XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63. August 2003.

Castagna, Giuseppe, Kim Nguyễn, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 289–302. January 2015.

Castagna, Giuseppe, Kim Nguyễn, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. Part 1: Syntax, semantics, and evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California, USA*, pages 5–17. January 2014.

Castagna, Giuseppe and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 94–106. September 2011.

Damas, Luis and Robin Milner. Principal type-schemes for functional programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico, USA*, pages 207–212. January 1982.

Frisch, Alain. *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. PhD thesis. Université Paris 7 – Denis Diderot, December 2004.

Frisch, Alain. OCaml + XDuce. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, Oregon, USA*, pages 192–200. September 2006.

Frisch, Alain, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–67, September 2008.

Garrigue, Jacques. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML, Baltimore, Maryland, USA*, informal proceedings. September 1998.

Garrigue, Jacques. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE), Sasaguri, Japan*, November 2000.

Garrigue, Jacques. Simple type inference for structural polymorphism. In *International Workshop on Foundations of Object-Oriented Languages (FOOL), Portland, Oregon, USA*, informal proceedings. January 2002.

Garrigue, Jacques. Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages, Gamagori, Japan*, March 2004.

Garrigue, Jacques. A certified implementation of ML with structural polymorphism and recursive types. *Mathematical Structures in Computer Science*, 25:867–891, May 2015.

Girard, Jean-Yves. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état. Université Paris 7, 1972. Summary in *Proceedings of the Second Scandinavian Logic Symposium* (J. E. Fenstad, editor), North-Holland, 1971 (pages 63–92).

Gordon, Michael J.C., Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

Hosoya, Haruo and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

Leroy, Xavier. *Typage polymorphe d'un langage algorithmique*. PhD thesis. Université Paris 7, June 1992.

Leroy, Xavier. The OCaml system release 4.02: Documentation and user's manual. With Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from ‹*http://caml.inria.fr*›. September 2014.

Maranget, Luc. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

Odersky, Martin, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. Summary in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, informal proceedings, 1997.

Ohori, Atsushi. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

Pierce, Benjamin C. *Types and Programming Languages*. MIT Press, 2002.

Pottier, François and Didier Rémy. The essence of ML type inference. Draft of an extended version. September 2003.

Pottier, François and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

Rémy, Didier. Type systems for programming languages. Course notes for the academic year 2013–2014. November 2013.

Tobin-Hochstadt, Sam and Matthias Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California, USA*, pages 395–406. January 2008.

Vytiniotis, Dimitrios, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(*X*) – Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4–5):333–412, September 2011.

Wand, Mitchell. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.