1 *ML*

By François Pottier and Didier Rémy

1.1 Preliminaries

Names and renaming

Mathematicians and computer scientists use *names* to refer to arbitrary or unknown objects in the statement of a theorem, to refer to the parameters of a function, *etc.* Names are convenient because they are understandable by humans; nevertheless, they can be tricky. An in-depth treatment of the difficulties associated with names and renaming is beyond the scope of the present chapter: we encourage the reader to study Gabbay and Pitts' excellent series of papers (Gabbay and Pitts, 2002; Pitts, 2002b). Here, we merely recall a few notions that are used throughout this chapter. Consider, for instance, an inductive definition of the abstract syntax of a simple programming language, the pure λ-calculus:

$$\mathtt{t} ::= \mathtt{z} \mid \lambda\mathtt{z}.\mathtt{t} \mid \mathtt{t}\ \mathtt{t}$$

Here, the *meta-variable* $\mathtt{z}$ ranges over an infinite set of *variables*—that is, names—while the meta-variable $\mathtt{t}$ ranges over *terms*. As usual in mathematics, we write "the variable $\mathtt{z}$" and "the term $\mathtt{t}$" instead of "the variable denoted by $\mathtt{z}$" and "the term denoted by $\mathtt{t}$". The above definition states that a term may be a variable $\mathtt{z}$, a pair of a variable and a term, written $\lambda\mathtt{z}.\mathtt{t}$, or a pair of terms, written $\mathtt{t}_1\ \mathtt{t}_2$. However, this is not quite what we need. Indeed, according to

The (currently unfinished) code that accompanies this chapter may be found at `http://pauillac.inria.fr/~remy/mlrow/`. For space reasons, some material, including proofs, exercises, and more, has been left out of this version. In the future, a full version of this chapter that includes the missing material will be made available at the same address. In spite of these omissions, this chapter is still oversize with respect to Benjamin's 100 page barrier: we currently have roughly 135 pages of text and 15 pages of solutions to exercises. We would appreciate comments and suggestions from the proofreaders as to how this chapter could be made shorter, without severely compromising its interest or readability.

this definition, the terms $\lambda \mathtt{z}_1.\mathtt{z}_1$ and $\lambda \mathtt{z}_2.\mathtt{z}_2$ are distinct, while we would like them to be a single mathematical object, because we intend $\lambda \mathtt{z}.\mathtt{z}$ to mean "the function that maps $\mathtt{z}$ to $\mathtt{z}$"—a meaning that is independent of the name $\mathtt{z}$. To achieve this effect, we complete the above definition by stating that the construction $\lambda \mathtt{z}.\mathtt{t}$ *binds* $\mathtt{z}$ within $\mathtt{t}$. One may also say that $\lambda \mathtt{z}$ is a *binder* whose *scope* is $\mathtt{t}$. Then, $\lambda \mathtt{z}.\mathtt{t}$ is no longer a pair: rather, it is an *abstraction* of the variable $\mathtt{z}$ within the term $\mathtt{t}$. Abstractions have the property that the identity of the bound variable does not matter; that is, $\lambda \mathtt{z}_1.\mathtt{z}_1$ and $\lambda \mathtt{z}_2.\mathtt{z}_2$ are the same term. Informally, we say that terms are considered equal modulo *α-conversion*. Once the position and scope of binders are known, several standard notions follow, such as the set of *free variables* of a term $\mathtt{t}$, written $\mathit{fv}(\mathtt{t})$, and the *capture-avoiding substitution* of a term $\mathtt{t}_1$ for a variable $\mathtt{z}$ within a term $\mathtt{t}_2$, written $[\mathtt{z} \mapsto \mathtt{t}_1]\mathtt{t}_2$. For conciseness, we write $\mathit{fv}(\mathtt{t}_1, \mathtt{t}_2)$ for $\mathit{fv}(\mathtt{t}_1) \cup \mathit{fv}(\mathtt{t}_2)$. A term is said to be *closed* when it has no free variables.

A *renaming* is a total bijective mapping from variables to variables that affects only a finite number of variables. The sole property of a variable is its identity, that is, the fact that it is distinct from other variables. As a result, at a global level, all variables are interchangeable: if a theorem holds in the absence of hypotheses about any particular variable, then any renaming of it holds as well. We often make use of this fact. When proving a theorem T, we say that a hypothesis H may be assumed *wihout loss of generality* (*w.l.o.g.*) if the theorem T follows from the theorem $H \Rightarrow T$ via a renaming argument, which is usually left implicit.

If $\bar{\mathtt{z}}_1$ and $\bar{\mathtt{z}}_2$ are sets of variables, we write $\bar{\mathtt{z}}_1 \mathrel{\#} \bar{\mathtt{z}}_2$ as a shorthand for $\bar{\mathtt{z}}_1 \cap \bar{\mathtt{z}}_2 = \varnothing$, and say that $\bar{\mathtt{z}}_1$ is *fresh* for $\bar{\mathtt{z}}_2$ (or vice-versa). We say that $\bar{\mathtt{z}}$ is fresh for $\mathtt{t}$ if and only if $\bar{\mathtt{z}} \mathrel{\#} \mathrm{fv}(\mathtt{t})$ holds.

In this chapter, we work with several distinct varieties of names: program variables, memory locations, and type variables, the latter of which may be further divided into *kinds*. We draw names of different varieties from disjoint sets, each of which is infinite.

1.2 What is ML?

The name "ML" appeared during the late seventies. It then referred to a general-purpose programming language that was used as a meta-language (whence its name) within the theorem prover LCF (Gordon, Milner, and Wadsworth, 1979b). Since then, several new programming languages, each of which offers several different implementations, have drawn inspiration from it. So, what does "ML" stand for today?

For a semanticist, "ML" might stand for a programming language featuring first-class functions, data structures built out of products and sums, mutable memory cells called *references*, exception handling, automatic memory management, and a call-by-value semantics. This view encompasses the Standard ML (Milner, Tofte, and Harper, 1990) and Caml (Leroy, 2000) families of programming languages. We refer to it as *ML-the-programming-language.*

For a type theorist, “ML” might stand for a particular breed of type systems, based on the simply-typed λ-calculus, but extended with a simple form of polymorphism introduced by `let` declarations. These type systems have decidable type inference; their type inference algorithms crucially rely on first-order unification and can be made efficient in practice. In addition to Standard ML and Caml, this view encompasses programming languages such as Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992) and Clean (Brus, van Eekelen, van Leer, and Plasmeijer, 1987), whose semantics is rather different—indeed, it is pure and lazy—but whose type system fits this description. We refer to it as *ML-the-type-system.* It is also referred to as *Hindley and Milner’s type discipline* in the literature.

For us, “ML” might also stand for the particular programming language whose formal definition is given and studied in this chapter. It is a core calculus featuring first-class functions, `let` declarations, and constants. It is equipped with a call-by-value semantics. By customizing constants and their semantics, one may recover data structures, references, and more. We refer to this particular calculus as *ML-the-calculus.*

Why study ML-the-type-system today, such a long time after its initial discovery? One may think of at least two reasons.

First, its treatment in the literature is often cursory, because it is considered either as a simple extension of the simply-typed λ-calculus (TAPL Chapter 9) or as a subset of Girard and Reynolds’ System F (TAPL Chapter 23). The former view is supported by the claim that the `let` construct, which distinguishes ML-the-type-system from the simply-typed λ-calculus, may be understood as a simple textual expansion facility. However, this view only tells part of the story, because it fails to give an account of the *principal types* property enjoyed by ML-the-type-system, leads to a naïve type inference algorithm whose time complexity is exponential, and breaks down when the language is extended with side effects, such as state or exceptions. The latter view is supported by the fact that every type derivation within ML-the-type-system is also a valid type derivation within an implicity-typed variant of System F. Such a view is correct, but again fails to give an account of type inference for ML-the-type-system, since type inference for System F is undecidable (Wells, 1999).

Second, existing accounts of type inference for ML-the-type-system (Milner, 1978; Damas and Milner, 1982; Tofte, 1988; Leroy, 1992; Lee and Yi, 1998;

Jones, 1999) usually involve heavy manipulations of type substitutions. Such an ubiquitous use of type substitutions is often quite obscure. Furthermore, actual implementations of the type inference algorithm do *not* explicitly manipulate substitutions; instead, they extend a standard first-order unification algorithm, where terms are updated in place as new equations are discovered (Huet, 1976). Thus, it is hard to tell, from these accounts, how to write an efficient type inference algorithm for ML-the-type-system. Yet, in spite of the increasing speed of computers, efficiency remains crucial when ML-the-type-system is extended with expensive features, such as Objective Caml's object types (Rémy and Vouillon, 1998) or polymorphic methods (Garrigue and Rémy, 1999).

For these reasons, we believe it is worth giving an account of ML-the-type-system that focuses on *type inference* and strives to be at once *elegant* and *faithful* to an efficient implementation. To achieve these goals, we forego type substitutions and instead put emphasis on *constraints*, which offer a number of advantanges. First, constraints allow a modular presentation of type inference as the combination of a constraint generator and a constraint solver. Such a decomposition allows reasoning separately about *when* a program is correct, on the one hand, and *how* to check whether it is correct, on the other hand. It has long been standard in the setting of the simply-typed λ-calculus (TAPL Chapter 22), but, to the best of our knowledge, has never been proposed for ML-the-type-system. Second, it is often natural to define and implement the solver as a constraint rewriting system. Then, the constraint language allows reasoning not only about correctness—is every rewriting step meaning-preserving?—but also about low-level implementation details, since constraints *are* the data structures manipulated throughout the type inference process. For instance, describing unification in terms of *multi-equations* (Jouannaud and Kirchner, 1991) allows reasoning about the sharing of nodes in memory, which a substitution-based approach cannot account for. Last, constraints are more general than type substitutions, and allow describing many extensions of ML-the-type-system, among which extensions with recursive types, rows, subtyping, first-order unification under a mixed prefix, and more.

Before delving into the details of this new presentation of ML-the-type-system, however, it is worth recalling its standard definition. Thus, in what follows, we first define the syntax and operational semantics of the programming language ML-the-calculus, and equip it with a type system, known as *Damas and Milner's type system*.

$\mathtt{x}, \mathtt{y}$	$::=$		*Identifiers:*
		$\mathtt{z}$	*Variable*
		m	*Memory location*
		$\mathtt{c}$	*Constant*
$\mathtt{t}$	$::=$		*Expressions:*
		$\mathtt{x}$	*Identifier*
		$\lambda\mathtt{z}.\mathtt{t}$	*Function*
		$\mathtt{t}\ \mathtt{t}$	*Application*
		$\mathtt{let}\ \mathtt{z} = \mathtt{t}\ \mathtt{in}\ \mathtt{t}$	*Local definition*
$\mathtt{v}, \mathtt{w}$	$::=$		*Values:*
		$\mathtt{z}$	*Variable*
		m	*Memory location*
		$\lambda\mathtt{z}.\mathtt{t}$	*Function*
		$\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$	*Data*
			$\mathtt{c} \in \mathcal{Q}^+ \wedge k \leq a(\mathtt{c})$
		$\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$	*Partial application*
			$\mathtt{c} \in \mathcal{Q}^- \wedge k < a(\mathtt{c})$
$\mathcal{E}$	$::=$		*Evaluation Contexts:*
		$[]$	*Empty context*
		$\mathcal{E}\ \mathtt{t}$	*Left side of an application*
		$\mathtt{v}\ \mathcal{E}$	*Right side of an application*
		$\mathtt{let}\ \mathtt{z} = \mathcal{E}\ \mathtt{in}\ \mathtt{t}$	*Local definition*

Figure 1-1: Syntax of ML-the-calculus

ML-the-calculus

The syntax of ML-the-calculus is defined in Figure 1-1. It is made up of several syntactic categories.

Identifiers group several kinds of names that may be referenced in a program: variables, memory locations, and constants. We let `x` and `y` range over identifiers. *Variables*—sometimes called *program variables* to avoid ambiguity—are names that may be bound to values using λ or `let` binding forms; in other words, they are names for function parameters or local definitions. We let `z` and `f` range over program variables. We sometimes write _ for a program variable that does not occur free within its scope: for instance, $\lambda\_.\mathtt{t}$ stands for $\lambda\mathtt{z}.\mathtt{t}$, provided `z` is fresh for `t`. *Memory locations* are names that represent memory addresses. By convention, memory locations never appear in source programs, that is, programs that are submitted to a compiler. They only appear during execution, when new memory blocks are allocated. *Constants* are fixed names for primitive values and operations, such as integer literals and integer arithmetic operations. Constants are elements of a finite or infinite set $\mathcal{Q}$. They are never subject to α-conversion. Program variables, memory locations, and constants belong to distinct syntactic classes and may never be confused.

The set of constants $\mathcal{Q}$ is kept abstract, so most of our development is independent of its concrete definition. We assume that every constant `c` has a nonnegative integer *arity* $a(\mathtt{c})$. We further assume that $\mathcal{Q}$ is partitioned into subsets of *constructors* $\mathcal{Q}^+$ and *destructors* $\mathcal{Q}^-$. Constructors and destructors differ in that the former are used to *form* values, while the latter are used to *operate* on values.

1.2.1 EXAMPLE [INTEGERS]: For every integer n, one may introduce a nullary constructor $\hat{n}$. In addition, one may introduce a binary destructor $\hat{+}$, whose applications are written infix, so $\mathtt{t}_1 \mathbin{\hat{+}} \mathtt{t}_2$ stands for the double application $\hat{+}\ \mathtt{t}_1\ \mathtt{t}_2$ of the destructor $\hat{+}$ to the expressions $\mathtt{t}_1$ and $\mathtt{t}_2$. □

Expressions—also known as *program terms* or *programs*—are the main syntactic category. Indeed, unlike procedural languages such as C and Java, functional languages, including ML-the-programming-language, suppress the distinction between expressions and statements. Expressions include identifiers, λ-abstractions, applications, and local definitions. The *λ-abstraction* $\lambda\mathtt{z}.\mathtt{t}$ represents the function of one parameter named $\mathtt{z}$ whose result is the expression $\mathtt{t}$, or, in other words, the function that maps $\mathtt{z}$ to $\mathtt{t}$. Note that the variable $\mathtt{z}$ is bound within the term $\mathtt{t}$, so (for instance) $\lambda\mathtt{z}_1.\mathtt{z}_1$ and $\lambda\mathtt{z}_2.\mathtt{z}_2$ are the same object. The *application* $\mathtt{t}_1\ \mathtt{t}_2$ represents the result of calling the function $\mathtt{t}_1$ with actual parameter $\mathtt{t}_2$, or, in other words, the result of applying $\mathtt{t}_1$ to $\mathtt{t}_2$. Application is left-associative, that is, $\mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3$ stands for $(\mathtt{t}_1\ \mathtt{t}_2)\ \mathtt{t}_3$. The construct $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ represents the result of evaluating $\mathtt{t}_2$ after binding the variable $\mathtt{z}$ to $\mathtt{t}_1$. Note that the variable $\mathtt{z}$ is bound within $\mathtt{t}_2$, but not within $\mathtt{t}_1$, so for instance $\mathtt{let}\ \mathtt{z}_1 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_1$ and $\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ are the same object. The construct $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ has the same meaning as $(\lambda\mathtt{z}.\mathtt{t}_2)\ \mathtt{t}_1$, but is dealt with in a more flexible way by ML-the-type-system. To sum up, the syntax of ML-the-calculus is that of the pure λ-calculus, extended with memory locations, constants, and the `let` construct.

Values form a subset of expressions. They are expressions whose evaluation is completed. Values include identifiers, λ-abstractions, and applications of constants, of the form $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, where k does not exceed $\mathtt{c}$'s arity if $\mathtt{c}$ is a constructor, and k is smaller than $\mathtt{c}$'s arity if $\mathtt{c}$ is a destructor. In what follows, we are often interested in closed values, that is, values that do not contain any free program variables. We use the meta-variables $\mathtt{v}$ and $\mathtt{w}$ for values.

1.2.2 EXAMPLE: The integer literals $\ldots, \widehat{-1}, \hat{0}, \hat{1}, \ldots$ are nullary constructors, so they are values. Integer addition $\hat{+}$ is a binary destructor, so it is a value, and so is every partial application $\hat{+}\ \mathtt{v}$. Thus, both $\hat{+}\ \hat{1}$ and $\hat{+}\ \hat{+}$ are values. An application of $\hat{+}$ to two values, such as $\hat{2}\hat{+}\hat{2}$, is not a value. □

1.2.3 EXAMPLE [PAIRS]: Let $(\cdot,\cdot)$ be a binary constructor. If $\mathtt{t}_1$ are $\mathtt{t}_2$ are expressions, then the double application $(\cdot,\cdot)\ \mathtt{t}_1\ \mathtt{t}_2$ may be called the *pair* of $\mathtt{t}_1$ and $\mathtt{t}_2$, and may be written $(\mathtt{t}_1,\mathtt{t}_2)$. By the definition above, $(\mathtt{t}_1,\mathtt{t}_2)$ is a value if and only if $\mathtt{t}_1$ and $\mathtt{t}_2$ are both values. □

Stores are finite mappings from memory locations to closed values. A store μ represents what is usually called a heap, that is, a collection of data structures, each of which is allocated at a particular address in memory and may contain pointers to other elements of the heap. ML-the-programming-language allows overwriting the contents of an existing memory block—an operation sometimes referred to as a *side effect*. In the operational semantics, this effect is achieved by mapping an existing memory location to a new value. We write $\varnothing$ for the empty store. We write $\mu[m \mapsto \mathtt{v}]$ for the store that maps m to $\mathtt{v}$ and otherwise coincides with μ. When μ and μ' have disjoint domains, we write $\mu\mu'$ for their union. We write $dom(\mu)$ for the domain of μ and $range(\mu)$ for the set of memory locations that appear in its codomain.

The operational semantics of a purely functional language, such as the pure λ-calculus, may be defined as a rewriting system on expressions. Because ML-the-calculus has side effects, however, we define its operational semantics as a rewriting system on *configurations*. A configuration $\mathtt{t}/\mu$ is a pair of an expression $\mathtt{t}$ and a store μ. The memory locations in the domain of μ are considered bound within $\mathtt{t}/\mu$, so (for instance) $m_1/(m_1 \mapsto \hat{0})$ and $m_2/(m_2 \mapsto \hat{0})$ are the same object. In what follows, we are often interested in *closed* configurations, that is, configurations $\mathtt{t}/\mu$ such that $\mathtt{t}$ has no free program variables and every memory location that appears within $\mathtt{t}$ or within the range of μ is in the domain of μ. If $\mathtt{t}$ is a source program, its evaluation begins within an empty store—that is, with the configuration $\mathtt{t}/\varnothing$. Because, by convention, source programs do not contain memory locations, this is a closed configuration. Furthermore, we shall see that all reducts of a closed configuration are closed as well. Please note that, instead of separating expressions and stores, it is possible to make store fragments part of the syntax of expressions; this idea, proposed in (Crank and Felleisen, 1991), is reminiscent of the encoding of reference cells in process calculi (Turner, 1995; Fournet and Gonthier, 1996).

A *context* is an expression where a single subexpression has been replaced with a *hole*, written $[]$. *Evaluation contexts* form a strict subset of contexts. In an evaluation context, the hole is meant to highlight a point in the program where it is valid to apply a reduction rule. Thus, the definition of evaluation contexts determines a reduction strategy: it tells where and in what order reduction steps may occur. For instance, the fact that $\lambda\mathtt{z}.[]$ is not an evaluation context means that the body of a function is never evaluated—that is, not until the function is applied, see R-BETA below. The fact that $\mathtt{t}\ \mathcal{E}$ is an evaluation context only if $\mathtt{t}$ is a value means that, to evaluate an application $\mathtt{t}_1\ \mathtt{t}_2$, one should fully evaluate $\mathtt{t}_1$ before attempting to evaluate $\mathtt{t}_2$. More generally, in the case of a multiple application, it means that arguments should be evaluated from left to right. Of course, other choices could be made: for instance, defining $\mathcal{E} ::= \ldots \mid \mathtt{t}\ \mathcal{E} \mid \mathcal{E}\ \mathtt{v} \mid \ldots$ would enforce a right-to-left evaluation order, while defining $\mathcal{E} ::= \ldots \mid \mathtt{t}\ \mathcal{E} \mid \mathcal{E}\ \mathtt{t} \mid \ldots$ would leave the evaluation order unspecified, effectively allowing reduction to alternate between

$$(\lambda \mathtt{z}.\mathtt{t})\ \mathtt{v} \longrightarrow [\mathtt{z} \mapsto \mathtt{v}]\mathtt{t} \qquad \text{(R-Beta)}$$

$$\mathtt{let}\ \mathtt{z} = \mathtt{v}\ \mathtt{in}\ \mathtt{t} \longrightarrow [\mathtt{z} \mapsto \mathtt{v}]\mathtt{t} \qquad \text{(R-Let)}$$

$$\frac{\mathtt{t}/\mu \xrightarrow{\delta} \mathtt{t}'/\mu'}{\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'} \qquad \text{(R-Delta)}$$

$$\frac{\begin{array}{c}\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu' \\ \mathit{dom}(\mu'') \ \#\ \mathit{dom}(\mu') \\ \mathit{range}(\mu'') \ \#\ \mathit{dom}(\mu' \setminus \mu)\end{array}}{\mathtt{t}/\mu\mu'' \longrightarrow \mathtt{t}'/\mu'\mu''} \qquad \text{(R-Extend)}$$

$$\frac{\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'}{\mathcal{E}[\mathtt{t}]/\mu \longrightarrow\!\!\!\!\triangleright\ \mathcal{E}[\mathtt{t}']/\mu'} \qquad \text{(R-Context)}$$

Figure 1-2: Semantics of ML-the-calculus

both subexpressions, and making evaluation nondeterministic. The fact that $\mathtt{let}\ \mathtt{z} = \mathtt{v}\ \mathtt{in}\ \mathcal{E}$ is not an evaluation context means that the body of a local definition is never evaluated—that is, not until the definition itself is reduced, see R-Let below. We write $\mathcal{E}[\mathtt{t}]$ for the expression obtained by replacing the hole in $\mathcal{E}$ with the expression $\mathtt{t}$.

Figure 1-2 defines first a relation $\longrightarrow$ between configurations, then a relation $\longrightarrow\!\!\!\!\triangleright$ between *closed* configurations. If $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$ or $\mathtt{t}/\mu \longrightarrow\!\!\!\!\triangleright\ \mathtt{t}'/\mu'$ holds, then we say that the configuration $\mathtt{t}/\mu$ *reduces* to the configuration $\mathtt{t}'/\mu'$; the ambiguity involved in this definition is benign. If $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu$ holds for every store μ, then we write $\mathtt{t} \longrightarrow \mathtt{t}'$ and say that the reduction is *pure*.

The key reduction rule is R-Beta, which states that a function application $(\lambda \mathtt{z}.\mathtt{t})\ \mathtt{v}$ reduces to the function body, namely $\mathtt{t}$, where every occurrence of the formal argument $\mathtt{z}$ has been replaced with the actual argument $\mathtt{v}$. The λ construct, which prevented the function body $\mathtt{t}$ from being evaluated, disappears, so the new term may (in general) be further reduced. Because ML-the-calculus adopts a *call-by-value* strategy, rule R-Beta is applicable only if the actual argument is a value $\mathtt{v}$. In other words, a function cannot be invoked until its actual argument has been fully evaluated. Rule R-Let is very similar to R-Beta. Indeed, it specifies that $\mathtt{let}\ \mathtt{z} = \mathtt{v}\ \mathtt{in}\ \mathtt{t}$ has the same behavior, with respect to reduction, as $(\lambda \mathtt{z}.\mathtt{t})\ \mathtt{v}$. We remark that substitution of a value for a program variable throughout a term is expensive, so R-Beta and R-Let are never implemented literally: they are only a simple *specification*. Actual implementations usually employ *runtime environments*, which may be understood as a form of *explicit substitutions* (Abadi, Cardelli, Curien, and Lévy, 1991). Please note that our choice of a call-by-value reduction strategy is fairly arbitrary, and has essentially no impact on the type system; the programming language Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992), whose reduction strategy is known as *lazy* or *call-by-need*, also relies on Hindley and Milner's type discipline.

Rule R-Delta describes the semantics of constants. It merely states that a certain relation $\xrightarrow{\delta}$ is a subset of $\longrightarrow$. Of course, since the set of constants is unspecified, the relation $\xrightarrow{\delta}$ must be kept abstract as well. We require that, if $\mathtt{t}/\mu \xrightarrow{\delta} \mathtt{t}'/\mu'$ holds, then

(i) $\mathtt{t}$ is of the form $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_n$, where $\mathtt{c}$ is a destructor of arity n; and

(ii) $dom(\mu)$ is a subset of $dom(\mu')$.

Condition (i) ensures that δ-reduction concerns full applications of destructors only, and that these are evaluated in accordance with the call-by-value strategy. Condition (ii) ensures that δ-reduction may allocate new memory locations, but not deallocate existing locations. In particular, a "garbage collection" operator, which destroys unreachable memory cells, cannot be made available as a constant. Doing so would not make much sense anyway in the presence of R-Extend, which states that any valid reduction is also valid in a larger store. Condition (ii) allows proving that, if $\mathtt{t}/\mu$ reduces to $\mathtt{t}'/\mu'$, then $dom(\mu)$ is a subset of $dom(\mu')$; this is left as an exercise to the reader.

1.2.4 Example [Integers, continued]: The operational semantics of integer addition may be defined as follows:

$$\hat{k}_1 \mathbin{\hat{+}} \hat{k}_2 \xrightarrow{\delta} \widehat{k_1 + k_2} \qquad \text{(R-Add)}$$

The left-hand term is the double application $\hat{+}\ \hat{k}_1\ \hat{k}_2$, while the right-hand term is the integer literal $\hat{k}$, where k is the sum of k_1 and k_2. The distinction between object level and meta level (that is, between $\hat{k}$ and k) is needed here to avoid ambiguity. □

1.2.5 Example [Pairs, continued]: In addition to the pair constructor defined in Example 1.2.3, we may introduce two destructors π_1 and π_2 of arity 1. We may define their operational semantics as follows, for $i \in \{1, 2\}$:

$$\pi_i\ \ (\mathtt{v}_1, \mathtt{v}_2) \xrightarrow{\delta} \mathtt{v}_i \qquad \text{(R-Proj)}$$

Thus, our treatment of constants is general enough to account for pair construction and destruction; we need not build these features explicitly into the language. □

1.2.6 Exercise [Booleans, Recommended, ★★]: Let `true` and `false` be nullary constructors. Let `if` be a ternary destructor. Extend the operational semantics with

$$\mathtt{if\ true}\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_1 \qquad \text{(R-True)}$$

$$\texttt{if false}\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_2 \qquad \text{(R-FALSE)}$$

Let us use the syntactic sugar `if` $\mathtt{t}_0$ `then` $\mathtt{t}_1$ `else` $\mathtt{t}_2$ for the triple application of `if` $\mathtt{t}_0\ \mathtt{t}_1\ \mathtt{t}_2$. Explain why these definitions do not quite provide the expected behavior. Without modifying the semantics of `if`, suggest a new definition of the syntactic sugar `if` $\mathtt{t}_0$ `then` $\mathtt{t}_1$ `else` $\mathtt{t}_2$ that corrects the problem. □

1.2.7 EXAMPLE [SUMS]: Booleans may in fact be viewed as a special case of the more general concept of *sum*. Let $\mathtt{inj}_1$ and $\mathtt{inj}_2$ be unary constructors, called respectively *left* and *right injections*. Let `case` be a ternary destructor, whose semantics is defined as follows, for $i \in \{1, 2\}$:

$$\mathtt{case}\ (\mathtt{inj}_i\ \mathtt{v})\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_i\ \mathtt{v} \qquad \text{(R-CASE)}$$

Here, the value $\mathtt{inj}_i\ \mathtt{v}$ is being scrutinized, while the values $\mathtt{v}_1$ and $\mathtt{v}_2$, which are typically functions, represent the two arms of a standard `case` construct. The rule selects an appropriate arm (here, $\mathtt{v}_i$) based on whether the value under scrutiny was formed using a left or right injection. The arm $\mathtt{v}_i$ is executed and given access to the data carried by the injection (here, `v`). □

1.2.8 EXERCISE [★, $\not\rightarrow$]: Explain how to encode `true`, `false` and the `if` construct in terms of sums. Check that the behavior of R-TRUE and R-FALSE is properly emulated. □

1.2.9 EXAMPLE [REFERENCES]: Let `ref` and ! be unary destructors. Let := be a binary destructor. We write $\mathtt{t}_1 := \mathtt{t}_2$ for the double application $:= \mathtt{t}_1\ \mathtt{t}_2$. Define the operational semantics of these three destructors as follows:

$$\mathtt{ref}\ \mathtt{v}/\varnothing \xrightarrow{\delta} m/(m \mapsto \mathtt{v}) \quad \text{if } m \text{ is fresh for } \mathtt{v} \qquad \text{(R-REF)}$$

$$!m/(m \mapsto \mathtt{v}) \xrightarrow{\delta} \mathtt{v}/(m \mapsto \mathtt{v}) \qquad \text{(R-DEREF)}$$

$$m := \mathtt{v}/(m \mapsto \mathtt{v}_0) \xrightarrow{\delta} \mathtt{v}/(m \mapsto \mathtt{v}) \qquad \text{(R-ASSIGN)}$$

According to R-REF, evaluating `ref v` allocates a fresh memory location m and binds `v` to it. Because configurations are considered equal up to α-conversion of memory locations, the choice of the name m is irrelevant, provided it is chosen fresh for `v`, so as to prevent inadvertent capture of the memory locations that appear free within `v`. By R-DEREF, evaluating $!m$ returns the value bound to the memory location m within the current store. By R-ASSIGN, evaluating $m := \mathtt{v}$ discards the value $\mathtt{v}_0$ currently bound to m and produces a new store where m is bound to `v`. Here, the value returned by the assignment $m := \mathtt{v}$ is `v` itself; in ML-the-programming-language, it is usually a nullary constructor (), pronounced *unit*. □

1.2.10 EXAMPLE [RECURSION]: Let `fix` be a binary destructor, whose operational semantics is:

$$\mathtt{fix}\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_1\ (\mathtt{fix}\ \mathtt{v}_1)\ \mathtt{v}_2 \qquad \text{(R-FIX)}$$

`fix` is a fixpoint combinator: it effectively allows recursive definitions of functions. Indeed, the construct $\mathtt{letrec}\ \mathtt{f} = \lambda\mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ provided by ML-the-programming-language may be viewed as syntactic sugar for $\mathtt{let}\ \mathtt{f} = \mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1)\ \mathtt{in}\ \mathtt{t}_2$. □

Rule R-CONTEXT completes the definition of the operational semantics by defining $\longrightarrow\!\!\triangleright$, a relation between closed configurations, in terms of $\xrightarrow{}$. The rule states that reduction may take place not only at the term's root, but also deep inside it, provided the path from the root to the point where reduction occurs forms an evaluation context. This is how evaluation contexts determine an evaluation strategy. As a purely technical point, because $\longrightarrow\!\!\triangleright$ relates closed configurations only, we do not need to require that the memory locations in $dom(\mu' \setminus \mu)$ be fresh for $\mathcal{E}$: indeed, every memory location that appears within $\mathcal{E}$ must be a member of $dom(\mu)$.

1.2.11 EXERCISE [★★, RECOMMENDED, $\not\twoheadrightarrow$]: Assuming the availability of Booleans and conditionals, integer literals, subtraction, multiplication, integer comparison, and a fixpoint combinator, most of which were defined in previous examples, define a function that computes the factorial of its integer argument, and apply it to $\hat{3}$. Determine, step by step, how this expression reduces to a value. □

It is straightforward to check that, if $\mathtt{t}/\mu$ reduces to $\mathtt{t}'/\mu'$, then `t` is not a value. In other words, values are irreducible: they represent a completed computation. The proof is left as an exercise to the reader. The converse, however, does not hold: if $\mathtt{t}/\mu$ is irreducible with respect to $\longrightarrow\!\!\triangleright$, then `t` is not necessarily a value. In that case, the configuration $\mathtt{t}/\mu$ is said to be *stuck*. It represents a *runtime error*, that is, a situation that does not allow computation to proceed, yet is not considered a valid outcome. A closed source program `t` is said to *go wrong* if and only if the configuration $\mathtt{t}/\varnothing$ reduces to a stuck configuration.

1.2.12 EXAMPLE: Runtime errors typically arise when destructors are applied to arguments of an unexpected nature. For instance, the expressions $\hat{+}\ 1\ m$ and $\pi_1\ \hat{2}$ and $!\hat{3}$ are stuck, regardless of the current store. The program $\mathtt{let}\ \mathtt{z} = \hat{+}\ \hat{+}\ \mathtt{in}\ \mathtt{z}\ \hat{1}$ is not stuck, because $\hat{+}\ \hat{+}$ is a value. However, its reduct through R-LET is $\hat{+}\ \hat{+}\ \hat{1}$, which is stuck, so this program goes wrong. The primary purpose of type systems is to prevent such situations from arising. □

1.2.13 EXAMPLE: The configuration $!m/\mu$ is stuck if m is not in the domain of μ. In that case, however, $!m/\mu$ is not closed. Because we consider $\longrightarrow\!\!\triangleright$ as a relation between closed configurations only, this situation cannot arise. In other words, the semantics of ML-the-calculus never allows the creation of *dangling pointers*. As a result, no particular precautions need be taken to guard against them. Several strongly typed programming languages do nevertheless allow dangling pointers in a controlled fashion (Tofte and Talpin, 1997; Crary, Walker, and Morrisett, 1999b; DeLine and Fähndrich, 2001; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney, 2002a). □

Damas and Milner's type system

ML-the-type-system was originally defined by Milner (1978). Here, we reproduce the definition given a few years later by Damas and Milner (1982), which is written in a more standard style: typing judgements are defined inductively by a collection of typing rules. We refer to this type system as DM.

To begin, we must define *types*. In DM, like in the simply-typed λ-calculus, types are first-order terms built out of *type constructors* and *type variables*. We begin with several considerations concerning the specification of type constructors.

First, we do not wish to fix the set of type constructors. Certainly, since ML-the-calculus has functions, we need to be able to form an arrow type $\mathtt{T} \rightarrow \mathtt{T}'$ out of arbitrary types $\mathtt{T}$ and $\mathtt{T}'$; that is, we need a binary type constructor $\rightarrow$. However, because ML-the-calculus includes an unspecified set of constants, we cannot say much else in general. If constants include integer literals and integer operations, as in Example 1.2.1, then a nullary type constructor `int` is needed; if they include pair construction and destruction, as in Examples 1.2.3 and 1.2.5, then a binary type constructor $\times$ is needed; and so on.

Second, it is common to refer to the parameters of a type constructor *by position*, that is, by numeric index. For instance, when one writes $\mathtt{T} \rightarrow \mathtt{T}'$, it is understood that the type constructor $\rightarrow$ has arity 2, that $\mathtt{T}$ is its *first* parameter, known as its *domain*, and that $\mathtt{T}'$ is its *second* parameter, known as its *codomain*. Here, however, we refer to parameters *by names*, known as *directions*. For instance, we define two directions *domain* and *codomain* and let the type constructor $\rightarrow$ have arity $\{domain, codomain\}$. The extra generality afforded by directions is exploited in the definition of nonstructural subtyping (Example 1.3.9) and in the definition of rows (Section 1.11).

Last, we allow types to be classified using *kinds*. As a result, every type constructor must come not only with an arity, but with a richer *signature*, which describes the kinds of the types to which it is applicable and the kind of the type that it produces. A distinguished kind $\star$ is associated with "normal" types, that is, types that are directly ascribed to expressions and values. For instance, the signature of the type constructor $\rightarrow$ is $\{domain \mapsto \star, codomain \mapsto \star\} \Rightarrow \star$, because it is applicable to two "normal" types and produces a "normal" type.

Introducing kinds other than $\star$ allows viewing some terms as ill-formed types; this is illustrated, for instance, in Section 1.11. In the simplest case, however, $\star$ is really the only kind, so the signature of a type constructor is nothing but its arity (a set of directions), and every term is a well-formed type, provided every application of a type constructor respects its arity.

1.2.14 Definition: Let d range over a finite or denumerable set of *directions*. Let κ range over a finite or denumerable set of *kinds*. Let $\star$ be a distinguished kind. Let K range over partial mappings from directions to kinds. Let F range over a finite or denumerable set of *type constructors*, each of which has a *signature* of the form $K \Rightarrow \kappa$. The domain of K is referred to as the *arity* of F, while κ is referred to as its *image kind*. We write κ instead of $K \Rightarrow \kappa$ when K is empty. Let $\rightarrow$ be a type constructor of signature $\{\mathit{domain} \mapsto \star, \mathit{codomain} \mapsto \star\} \Rightarrow \star$. $\square$

The type constructors and their signatures collectively form a *signature* $\mathcal{S}$. In the following, we assume that a fixed signature $\mathcal{S}$ is given and that every type constructor in it has *finite* arity, so as to ensure that types are machine representable. However, in Section 1.11, we shall explicitly work with several distinct signatures, some of which involve type constructors of denumerable arity.

A *type variable* is a name that is used to stand for a type. For simplicity, we assume that every type variable is branded with a kind, or, in other words, that type variables of distinct kinds are drawn from disjoint sets. Each of these sets of type variables is individually subject to α-conversion: that is, renamings must preserve kinds. Attaching kinds to type variables is only a technical convenience: in practice, every operation performed during type inference preserves the property that every type is well-kinded, so it is not necessary to keep track of the kind of every type variable. It is only necessary to check that all types supplied by the user, within type declarations, type annotations, or module interfaces, are well-kinded.

1.2.15 Definition: For every kind κ, let $\mathcal{V}_\kappa$ be a disjoint, denumerable set of *type variables*. Let $\mathtt{X}$, $\mathtt{Y}$, and $\mathtt{Z}$ range over the set $\mathcal{V}$ of all type variables. Let $\bar{\mathtt{X}}$ and $\bar{\mathtt{Y}}$ range over finite sets of type variables. We write $\bar{\mathtt{X}}\bar{\mathtt{Y}}$ for the set $\bar{\mathtt{X}} \cup \bar{\mathtt{Y}}$ and often write $\mathtt{X}$ for the singleton set $\{\mathtt{X}\}$. We write $\mathit{ftv}(o)$ for the set of *free type variables* of an object o. $\square$

The set of types, ranged over by $\mathtt{T}$, is the free many-kinded term algebra that arises out of the type constructors and type variables.

1.2.16 Definition: A *type* of kind κ is either a member of $\mathcal{V}_\kappa$, or a term of the form $F\,\{d_1 \mapsto \mathtt{T}_1, \ldots, d_n \mapsto \mathtt{T}_n\}$, where F has signature $\{d_1 \mapsto \kappa_1, \ldots, d_n \mapsto \kappa_n\} \Rightarrow \kappa$ and $\mathtt{T}_1, \ldots, \mathtt{T}_n$ are types of kind $\kappa_1, \ldots, \kappa_n$, respectively. $\square$

As a notational convention, we assume that, for every type constructor F, the directions that form the arity of F are implicitly ordered, so that we may say that F has signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \kappa$ and employ the syntax $F\,\mathtt{T}_1\,\ldots\,\mathtt{T}_n$ for applications of F. Applications of the type constructor $\rightarrow$ are written infix and associate to the right, so $\mathtt{T} \rightarrow \mathtt{T}' \rightarrow \mathtt{T}''$ stands for $\mathtt{T} \rightarrow (\mathtt{T}' \rightarrow \mathtt{T}'')$.

In order to give meaning to the free type variables of a type, or, more generally, of a typing judgement, traditional presentations of ML-the-type-system, including Damas and Milner's, employ *type substitutions*. Most of our presentation avoids substitutions and uses *constraints* instead. However, we do need substitutions on a few occasions, especially when relating our presentation to Damas and Milner's.

1.2.17 Definition: A *type substitution* θ is a total, kind-preserving mapping of type variables to types that is the identity everywhere but on a finite subset of $\mathcal{V}$, which we call the *domain* of θ and write $dom(\theta)$. The *range* of θ, which we write $range(\theta)$, is the set $ftv(\theta(dom(\theta)))$. A type substitution may naturally be viewed as a total, kind-preserving mapping of types to types. In the following, we write $\bar{\mathtt{X}} \# \theta$ for $\bar{\mathtt{X}} \# (dom(\theta) \cup range(\theta))$. We write $\theta \setminus \bar{\mathtt{X}}$ for the restriction of θ outside $\bar{\mathtt{X}}$, that is, the restriction of θ to $\mathcal{V} \setminus \bar{\mathtt{X}}$. We sometimes let φ denote a type substitution. □

If $\vec{\mathtt{X}}$ and $\vec{\mathtt{T}}$ are respectively a vector of distinct type variables and a vector of types of the same (finite) length, such that, for every index i, $\mathtt{X}_i$ and $\mathtt{T}_i$ have the same kind, then $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ denotes the substitution that maps $\mathtt{X}_i$ to $\mathtt{T}_i$ for every index i. The domain of $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ is a subset of $\bar{\mathtt{X}}$, the set underlying the vector $\vec{\mathtt{X}}$. Its range is a subset of $ftv(\bar{\mathtt{T}})$, where $\bar{\mathtt{T}}$ is the set underlying the vector $\vec{\mathtt{T}}$. Every substitution θ may be written under the form $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$, where $\bar{\mathtt{X}} = dom(\theta)$. Then, θ is *idempotent* if and only if $\bar{\mathtt{X}} \# ftv(\bar{\mathtt{T}})$ holds.

As pointed out earlier, types are first-order terms; that is, in the grammar of types, none of the productions *binds* a type variable. As a result, every type variable that appears within a type $\mathtt{T}$ appears *free* within $\mathtt{T}$. This situation is identical to that of the simply-typed λ-calculus. Things become more interesting when we introduce *type schemes*. As its name implies, a type scheme may describe an entire family of types; this effect is achieved via *universal quantification* over a set of type variables.

1.2.18 Definition: A type scheme S is an object of the form $\forall\bar{\mathtt{X}}.\mathtt{T}$, where $\mathtt{T}$ is a type of kind $\star$ and the type variables $\bar{\mathtt{X}}$ are considered bound within $\mathtt{T}$. □

One may view the type $\mathtt{T}$ as the trivial type scheme $\forall\varnothing.\mathtt{T}$, where no type variables are universally quantified, so types may be viewed as a subset of type schemes. The type scheme $\forall\bar{\mathtt{X}}.\mathtt{T}$ may be viewed as a finite way of describing the possibly infinite family of types of the form $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$, where $\vec{\mathtt{T}}$ is arbitrary.

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}}{\Gamma \vdash \mathtt{x} : \mathtt{S}} \quad (\textsc{dm-Var}) \qquad \frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma; \mathtt{z} : \mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}} \quad (\textsc{dm-Let})$$

$$\frac{\Gamma; \mathtt{z} : \mathtt{T} \vdash \mathtt{t} : \mathtt{T}'}{\Gamma \vdash \lambda \mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'} \quad (\textsc{dm-Abs}) \qquad \frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)}{\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}} \quad (\textsc{dm-Gen})$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}' \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'} \quad (\textsc{dm-App}) \qquad \frac{\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}}{\Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}} \quad (\textsc{dm-Inst})$$

Figure 1-3: Typing rules for DM

Such types are called *instances* of the type scheme $\forall\bar{\mathtt{X}}.\mathtt{T}$. Note that, throughout most of this chapter, we work with *constrained type schemes*, a generalization of DM type schemes (Definition 1.3.2).

Typing environments, or environments for short, are used to collect assumptions about an expression's free identifiers.

1.2.19 Definition: An *environment* Γ is a finite ordered sequence of pairs of a program identifier and a type scheme. We write $\varnothing$ for the empty environment and ; for the concatenation of environments. An environment may be viewed as a finite mapping from program identifiers to type schemes by letting $\Gamma(\mathtt{x}) = \mathtt{S}$ if and only if Γ is of the form $\Gamma_1; \mathtt{x} : \mathtt{S}; \Gamma_2$, where Γ_2 contains no assumption about $\mathtt{x}$. The set of *defined program identifiers* of an environment Γ, written $\mathit{dpi}(\Gamma)$, is defined by $\mathit{dpi}(\varnothing) = \varnothing$ and $\mathit{dpi}(\Gamma; \mathtt{x} : \mathtt{S}) = \mathit{dpi}(\Gamma) \cup \{\mathtt{x}\}$. □

To complete the definition of Damas and Milner's type system, there remains to define *typing judgements*. A typing judgement takes the form $\Gamma \vdash \mathtt{t} : \mathtt{S}$, where $\mathtt{t}$ is an expression of interest, Γ is an environment, which typically contains assumptions about $\mathtt{t}$'s free program identifiers, and $\mathtt{S}$ is a type scheme. Such a judgement may be read: *under assumptions* Γ*, the expression* $\mathtt{t}$ *has the type scheme* $\mathtt{S}$. By abuse of language, it is sometimes said that $\mathtt{t}$ *has type* $\mathtt{S}$. A typing judgement is *valid* (or *holds*) if and only if it may be derived using the rules that appear in Figure 1-3. An expression $\mathtt{t}$ is *well-typed* within the environment Γ if and only if some judgement of the form $\Gamma \vdash \mathtt{t} : \mathtt{S}$ holds; it is *ill-typed* within Γ otherwise.

Rule dm-Var allows fetching a type scheme for an identifier $\mathtt{x}$ from the environment. It is equally applicable to program variables, memory locations, and constants. If no assumption concerning $\mathtt{x}$ appears in the environment Γ, then the rule isn't applicable. In that case, the expression $\mathtt{x}$ is ill-typed within Γ—can you prove it? Assumptions about constants are usually collected in a so-called *initial environment* Γ_0. It is the environment under which closed programs are typechecked, so every subexpression is typechecked under some extension Γ of Γ_0. Of course, the type schemes assigned by Γ_0 to constants must be consistent with their operational semantics; we say more about this later (Section 1.7). Rule DM-ABS specifies how to typecheck a λ-abstraction $\lambda\mathtt{z}.\mathtt{t}$. Its premise requires the body of the function, namely $\mathtt{t}$, to be well-typed under an extra assumption, which causes all free occurrences of $\mathtt{z}$ within $\mathtt{t}$ to receive a common type $\mathtt{T}$. Its conclusion forms the arrow type $\mathtt{T} \rightarrow \mathtt{T}'$ out of the types of the function's formal parameter, namely $\mathtt{T}$, and result, namely $\mathtt{T}'$. It is worth noting that this rule always augments the environment with a type $\mathtt{T}$—recall that, by convention, types form a subset of type schemes—but never with a nontrivial type scheme. DM-APP states that the type of a function application is the codomain of the function's type, provided that the domain of the function's type is a valid type for the actual argument. DM-LET closely mirrors the operational semantics: whereas the semantics of the local definition $\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ is to augment the *runtime* environment by binding $\mathtt{z}$ to the *value* of $\mathtt{t}_1$ prior to evaluating $\mathtt{t}_2$, the effect of DM-LET is to augment the *typing* environment by binding $\mathtt{z}$ to a *type scheme* for $\mathtt{t}_1$ prior to typechecking $\mathtt{t}_2$. DM-GEN turns a type into a type scheme by universally quantifying over a set of type variables that do not appear free in the environment; this restriction is discussed in Example 1.2.20 below. DM-INST, on the contrary, turns a type scheme into one of its instances, which may be chosen arbitrarily. These two operations are referred to as *generalization* and *instantiation*. The notion of type scheme and the rules DM-GEN and DM-INST are characteristic of ML-the-type-system: they distinguish it from the simply-typed λ-calculus.

1.2.20 EXAMPLE: It is unsound to allow generalizing type variables that appear free in the environment. For instance, consider the typing judgement $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \mathtt{X}$ **(1)**, which, according to DM-VAR, is valid. Applying an unrestricted version of DM-GEN to it, we obtain $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \forall\mathtt{X}.\mathtt{X}$ **(2)**, whence, by DM-INST, $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \mathtt{Y}$ **(3)**. By DM-ABS and DM-GEN, we then have $\varnothing \vdash \lambda\mathtt{z}.\mathtt{z} : \forall\mathtt{XY}.\mathtt{X} \rightarrow \mathtt{Y}$. In other words, the identity function has unrelated argument and result types! Then, the expression $(\lambda\mathtt{z}.\mathtt{z})\ \hat{0}\ \hat{0}$, which reduces to the stuck expression $\hat{0}\ \hat{0}$, has type scheme $\forall\mathtt{Z}.\mathtt{Z}$. So, well-typed programs may cause runtime errors: the type system is unsound.

What happened? It is clear that the judgement (1) is correct only because the type assigned to $\mathtt{z}$ is the *same* in its assumption and in its right-hand side. For the same reason, the judgements (2) and (3)—the former of which may be written $\mathtt{z} : \mathtt{X} \vdash \mathtt{z} : \forall\mathtt{Y}.\mathtt{Y}$—are incorrect. Indeed, such judgements defeat the very purpose of environments, since they disregard their assumption.

By universally quantifying over $\mathtt{X}$ *in the right-hand side only*, we break the connection between occurrences of $\mathtt{X}$ in the assumption, which remain free, and occurrences in the right-hand side, which become bound. This is correct only if there are in fact no free occurrences of $\mathtt{X}$ in the assumption. □

It is a key feature of ML-the-type-system that DM-ABS may only introduce a type $\mathtt{T}$, rather than a type scheme, into the environment. Indeed, this allows the rule's conclusion to form the arrow type $\mathtt{T} \rightarrow \mathtt{T}'$. If instead the rule were to introduce the assumption $\mathtt{z} : \mathtt{S}$ into the environment, then its conclusion would have to form $\mathtt{S} \rightarrow \mathtt{T}'$, which is not a well-formed type. In other words, this restriction is necessary to preserve the stratification between types and type schemes. If we were to remove this stratification, thus allowing universal quantifiers to appear deep inside types, we would obtain an implicitly-typed version of System F (TAPL Chapter 23). Type inference for System F is undecidable (Wells, 1999), while type inference for ML-the-type-system is decidable, as we show later, so this design choice has a rather drastic impact.

1.2.21 EXERCISE [★, RECOMMENDED]: Build a type derivation for the expression $\lambda \mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ within DM. □

1.2.22 EXERCISE [★, RECOMMENDED]: Let $\mathtt{int}$ be a nullary type constructor of signature $\star$. Let Γ_0 consist of the bindings $\hat{+} : \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int}$ and $\hat{k} : \mathtt{int}$, for every integer k. Can you find derivations of the following valid typing judgements? Which of these judgements are valid in the simply-typed λ-calculus, where $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ is syntactic sugar for $(\lambda \mathtt{z}.\mathtt{t}_2)\ \mathtt{t}_1$?

$$\Gamma_0 \vdash \lambda \mathtt{z}.\mathtt{z} : \mathtt{int} \rightarrow \mathtt{int}$$
$$\Gamma_0 \vdash \lambda \mathtt{z}.\mathtt{z} : \forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$$
$$\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{z} \hat{+} \hat{1}\ \mathtt{in}\ \mathtt{f}\ \hat{2} : \mathtt{int}$$
$$\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{z}\ \mathtt{in}\ \mathtt{f}\ \mathtt{f}\ \hat{2} : \mathtt{int}$$

Show that the expressions $\hat{1}\ \hat{2}$ and $\lambda \mathtt{f}.(\mathtt{f}\ \mathtt{f})$ are ill-typed within Γ_0. Could these expressions be well-typed in a more powerful type system? □

1.2.23 EXERCISE [★★]: In fact, the rules shown in Figure 1-3 are not exactly Damas and Milner's original rules. In (Damas and Milner, 1982), the generalization and instantiation rules are:

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{S} \qquad \mathtt{X} \notin ftv(\Gamma)}{\Gamma \vdash \mathtt{t} : \forall \mathtt{X}.\mathtt{S}} \qquad \text{(DM-GEN')}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T} \qquad \bar{\mathtt{Y}} \# ftv(\forall \bar{\mathtt{X}}.\mathtt{T})}{\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{Y}}.[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}} \qquad \text{(DM-INST')}$$

where $\forall \texttt{X}.\texttt{S}$ stands for $\forall \texttt{X}\bar{\texttt{X}}.\texttt{T}$ if $\texttt{S}$ stands for $\forall \bar{\texttt{X}}.\texttt{T}$. Show that the combination of DM-GEN' and DM-INST' is equivalent to the combination of DM-GEN and DM-INST. □

DM enjoys a number of nice theoretical properties, which have practical implications. First, under suitable hypotheses about the semantics of constants, about the type schemes that they receive in the initial environment, and—in the presence of side effects—under a slight restriction of the syntax of `let` constructs, it is possible to show that the type system is sound: that is, *well-typed (closed) programs do not go wrong.* This essential property ensures that programs that are accepted by the typechecker may be compiled without runtime checks. Furthermore, it is possible to show that there exists an algorithm that, given a (closed) environment Γ and a program $\texttt{t}$, tells whether $\texttt{t}$ is well-typed with respect to Γ, and if so, produces a *principal* type scheme $\texttt{S}$. A principal type scheme is such that (i) it is valid, that is, $\Gamma \vdash \texttt{t} : \texttt{S}$ holds, and (ii) it is most general, that is, every judgement of the form $\Gamma \vdash \texttt{t} : \texttt{S}'$ follows from $\Gamma \vdash \texttt{t} : \texttt{S}$ by DM-INST and DM-GEN. (For the sake of simplicity, we have stated the properties of the type inference algorithm only in the case of a closed environment Γ; the specification is slightly heavier in the general case.) This implies that *type inference is decidable*: the compiler does not require expressions to be annotated with types. It also implies that, under a fixed environment Γ, all of the type information associated with an expression $\texttt{t}$ may be summarized in the form of a single (principal) type scheme, which is very convenient.

Road map

Before proving the above claims, we first generalize our presentation by moving to a *constraint-based* setting. The necessary tools, namely the constraint language, its interpretation, and a number of constraint equivalence laws, are introduced in Section 1.3. In Section 1.4, we describe the standard constraint-based type system HM(X) (Odersky, Sulzmann, and Wehr, 1999a; Sulzmann, Müller, and Zenger, 1999; Sulzmann, 2000). We prove that, when constraints are made up of equations between free, finite terms, HM(X) is a reformulation of DM. In the presence of a more powerful constraint language, HM(X) is an extension of DM. In Section 1.5, we propose an original reformulation of HM(X), dubbed PCB(X), whose distinctive feature is to exploit *type scheme introduction* and *instantiation* constraints. In Section 1.6, we show that, thanks to the extra expressive power afforded by these constraint forms, type inference may be viewed as a combination of constraint generation and constraint solving, as promised earlier. Indeed, we define a constraint generator and relate it with PCB(X). Then, in Section 1.7, we give a type soundness theorem. It is stated purely in terms of constraints, but—thanks to the results developed in the previous sections—applies equally to PCB(X), HM(X), and DM.

Throughout this core material, the syntax and interpretation of constraints are left partly unspecified. Thus, the development is *parameterized* with respect to them—hence the unknown X in the names HM(X) and PCB(X). We really describe a *family* of constraint-based type systems, all of which *share* a common constraint generator and a common type soundness proof. Constraint solving, however, cannot be independent of X: on the contrary, the design of an efficient solver is heavily dependent on the syntax and interpretation of constraints. In Section 1.8, we consider constraint solving in the particular case where constraints are made up of equations interpreted in a free tree model, and define a constraint solver on top of a standard first-order unification algorithm.

The remainder of this chapter deals with extensions of the framework. In Section 1.9, we explain how to extend ML-the-calculus with a number of features, including data structures, pattern matching, and type annotations. In Section 1.10, we extend the constraint language with universal quantification and describe a number of extra features that require this extension, including a different flavor of type annotations, polymorphic recursion, and first-class universal and existential types. Last, in Section 1.11, we extend the constraint language with *rows* and describe their applications, which include extensible variants and records.

1.3 Constraints

In this section, we define the syntax and logical meaning of constraints. Both are partly unspecified. Indeed, the set of *type constructors* (Definition 1.2.14) must contain at least the binary type constructor $\rightarrow$, but might contain more. Similarly, the syntax of constraints involves a set of so-called *predicates* on types, which we require to contain at least a binary *subtyping* predicate $\leq$, but might contain more. Furthermore, the logical interpretation of type constructors and of predicates is left almost entirely unspecified. This freedom allows reasoning not only about Damas and Milner's type system, but also about a family of constraint-based extensions of it.

Type constructors other than $\rightarrow$ and predicates other than $\leq$ will never explicitly appear in the definition of our constraint-based type systems, precisely because the definition is parametric with respect to them. They can (and usually do) appear in the type schemes assigned to constructors and destructors by the initial environment Γ_0.

The introduction of subtyping has little impact on the complexity of our

σ ::=		*type scheme:*
	$\forall \bar{\mathtt{X}}[C].\mathtt{T}$	
C, D ::=		*constraint:*
	true	*truth*
	false	*falsity*
	$P\,\mathtt{T}_1 \ldots \mathtt{T}_n$	*predicate application*
	$C \wedge C$	*conjunction*
	$\exists \bar{\mathtt{X}}.C$	*existential quantification*
	$\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$	*type scheme introduction*
	$\mathtt{x} \preceq \mathtt{T}$	*type scheme instantiation*
Γ ::=		*Typing environments:*
	$\varnothing$	
	$x : \sigma$	
	$\Gamma; \Gamma$	
C, D ::=		*Syntactic sugar for constraints:*
	$\ldots$	*As before*
	$\sigma \preceq \mathtt{T}$	*Definition 1.3.3*
	$\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$	*Definition 1.3.3*
	$\exists \sigma$	*Definition 1.3.3*
	$\mathsf{def}\ \Gamma\ \mathsf{in}\ C$	*Definition 1.3.4*
	$\mathsf{let}\ \Gamma\ \mathsf{in}\ C$	*Definition 1.3.4*
	$\exists \Gamma$	*Definition 1.3.4*

Figure 1-4: Syntax of type schemes and constraints

proofs, yet increases the framework's expressive power. When subtyping is not desired, we interpret the predicate $\leq$ as equality.

Syntax

We now define the syntax of constrained type schemes and of constraints, and introduce some extra constraint forms as syntactic sugar.

1.3.1 Definition: Let P range over a finite or denumerable set of *predicates*, each of which has a *signature* of the form $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, where $n \geq 0$. Let $\leq$ be a distinguished predicate of signature $\star \otimes \star \Rightarrow \cdot$. □

1.3.2 Definition: The syntax of *type schemes* and *constraints* is given in Figure 1-4. It is further restricted by the following requirements. In the type scheme $\forall \bar{\mathtt{X}}[C].\mathtt{T}$ and in the constraint $\mathtt{x} \preceq \mathtt{T}$, the type $\mathtt{T}$ must have kind $\star$. In the constraint $P\,\mathtt{T}_1 \ldots \mathtt{T}_n$, the types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ must have kind $\kappa_1, \ldots, \kappa_n$, respectively, if P has signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$. We write $\forall \bar{\mathtt{X}}.\mathtt{T}$ for $\forall \bar{\mathtt{X}}[\mathsf{true}].\mathtt{T}$, which allows viewing DM type schemes as a subset of constrained type schemes. □

We write $\mathtt{T}_1 \leq \mathtt{T}_2$ for the binary predicate application $\leq \mathtt{T}_1\,\mathtt{T}_2$, and call it a subtyping constraint. By convention, $\exists$ and def bind tighter than $\wedge$; that is, $\exists \bar{\mathtt{X}}.C \wedge D$ is $(\exists \bar{\mathtt{X}}.C) \wedge D$ and $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C \wedge D$ is $(\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C) \wedge D$. In $\forall \bar{\mathtt{X}}[C].\mathtt{T}$, the type variables $\bar{\mathtt{X}}$ are bound within C and $\mathtt{T}$. In $\exists \bar{\mathtt{X}}.C$, the type variables $\bar{\mathtt{X}}$ are bound within C. The sets of free type variables of a type scheme σ and of a constraint C, written $ftv(\sigma)$ and $ftv(C)$, respectively, are defined accordingly. In $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$, the identifier $\mathtt{x}$ is bound within C. The sets of free program identifiers of a type scheme σ and of a constraint C, written $\mathit{fpi}(\sigma)$ and $\mathit{fpi}(C)$, respectively, are defined accordingly. Please note that $\mathtt{x}$ occurs free in the constraint $\mathtt{x} \preceq \mathtt{T}$.

We immediately introduce a number of derived constraint forms:

1.3.3 Definition: Let σ be $\forall\bar{\mathtt{X}}[C].\mathtt{T}$. If $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}')$ holds, then $\sigma \preceq \mathtt{T}'$ (read: $\mathtt{T}'$ *is an instance of* σ) stands for the constraint $\exists\bar{\mathtt{X}}.(C \wedge \mathtt{T} \leq \mathtt{T}')$. We write $\exists\sigma$ (read: σ *has an instance*) for $\exists\bar{\mathtt{X}}.C$ and $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$ for $\exists\sigma \wedge \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$. □

Constrained type schemes generalize Damas and Milner's type schemes, while our definition of instantiation constraints generalizes Damas and Milner's instance relation (Definition 1.2.18). Let us draw a comparison. First, Damas and Milner's instance relation yields a "yes/no" answer, and is purely syntactic: for instance, the type $\mathtt{Y} \rightarrow \mathtt{Z}$ is *not* an instance of $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$ in Damas and Milner's sense, because $\mathtt{Y}$ and $\mathtt{Z}$ are distinct type variables. In our presentation, on the other hand, $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X} \preceq \mathtt{Y} \rightarrow \mathtt{Z}$ is not an assertion; rather, it is a constraint, which by definition is $\exists\mathtt{X}.(\mathsf{true} \wedge \mathtt{X} \rightarrow \mathtt{X} \leq \mathtt{Y} \rightarrow \mathtt{Z})$. We later prove that it is equivalent to $\exists\mathtt{X}.(\mathtt{Y} \leq \mathtt{X} \wedge \mathtt{X} \leq \mathtt{Z})$ and to $\mathtt{Y} \leq \mathtt{Z}$, or, if subtyping is interpreted as equality, to $\mathtt{Y} = \mathtt{Z}$. That is, $\sigma \preceq \mathtt{T}'$ represents a condition on (the types denoted by) the type variables in $\mathit{ftv}(\sigma, \mathtt{T}')$ for $\mathtt{T}'$ to be an instance of σ, in a logical, rather than purely syntactic, sense. Second, the definition of instantiation constraints involves subtyping, so as to ensure that any supertype of an instance of σ is again an instance of σ (see rule C-ExTrans in Figure 1-6 and Lemma 1.3.17). This is consistent with the purpose of subtyping, which is to allow supplying a subtype where a supertype is expected (TAPL Chapter 15). Third and last, every type scheme now carries a constraint. The constraint C, whose free type variables may or may not be members of $\bar{\mathtt{X}}$, restricts the instances of the type scheme $\forall\bar{\mathtt{X}}[C].\mathtt{T}$. This is expressed in the instantiation constraint $\exists\bar{\mathtt{X}}.(C \wedge \mathtt{T} \leq \mathtt{T}')$, where the values that $\bar{\mathtt{X}}$ may assume are restricted by the requirement that C be satisfied. This requirement vanishes in the case of DM type schemes, where C is true. Our notions of constrained type scheme and of instantiation constraint are standard: they are exactly those of HM(X) (Odersky, Sulzmann, and Wehr, 1999a).

The constraint true, which is always satisfied, mainly serves to indicate the absence of a nontrivial constraint, while false, which has no solution, may be understood as the indication of a type error. Composite constraints include conjunction and existential quantification, which have their standard meaning, as well as *type scheme introduction* and *type scheme instantiation* constraints, which are similar to Gustavsson and Svenningsson's constraint abstractions (2001b). In short, the construct $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$ binds the name $\mathtt{x}$ to the type scheme σ within the constraint C. If C contains a subconstraint of the form $\mathtt{x} \preceq \mathtt{T}$, where this occurrence of $\mathtt{x}$ is free in C, then this subconstraint acquires the meaning $\sigma \preceq \mathtt{T}$. Thus, the constraint $\mathtt{x} \preceq \mathtt{T}$ is indeed an instantiation constraint, where the type scheme that is being instantiated is referred to by name. The constraint $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$ may be viewed as an *explicit substitution* of the type scheme σ for the name $\mathtt{x}$ within C. Later (Section 1.5), we use such explicit substitutions to supplant typing environments. That is, where Damas and Milner's type system augments the current typing environment (DM-ABS, DM-LET), we introduce a new def binding in the current constraint; where it looks up the current typing environment (DM-VAR), we employ an instantiation constraint. The point is that it is then up to a constraint solver to choose a strategy for reducing explicit substitutions—for instance, one might wish to *simplify* σ before substituting it for $\mathtt{x}$ within C—whereas the use of environments in standard type systems such as DM and HM(X) imposes an eager substitution strategy, which is inefficient and thus never literally implemented. The use of type scheme introduction and instantiation constraints allows separating constraint generation and constraint solving *without compromising efficiency*, or, in other words, without introducing a gap between the description of the type inference algorithm and its actual implementation. Although the algorithm that we plan to describe is not new, its description in terms of constraints is: to the best of our knowledge, the only close relative of our def constraints is to be found in (Gustavsson and Svenningsson, 2001b). Fähndrich, Rehof, and Das's instantiation constraints (2000) are also related, but may be recursive and are meant to be solved using a semi-unification procedure, as opposed to a unification algorithm extended with facilities for creating and instantiating type schemes, as in our case.

One consequence of introducing constraints inside type schemes is that some type schemes have no instances at all, or have instances only if a certain constraint holds. For instance, the type scheme $\sigma = \forall\mathtt{X}[\mathtt{bool} = \mathtt{int}].\mathtt{X}$, where the nullary type constructors $\mathtt{int}$ and $\mathtt{bool}$ have distinct interpretations, has no instances; that is, no constraint of the form $\sigma \preceq \mathtt{T}'$ has a solution. The type scheme $\sigma = \forall\mathtt{Z}[\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Z}].\mathtt{Z}$ has an instance only if $\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Z}$ holds for some $\mathtt{Z}$; in other words, for every $\mathtt{T}'$, $\sigma \preceq \mathtt{T}'$ entails $\exists\mathtt{Z}.(\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Z})$. (We define *entailment* on page 29.) We later prove that the constraint $\exists\sigma$ is equivalent to $\exists\mathtt{Z}.\sigma \preceq \mathtt{Z}$, where $\mathtt{Z} \notin \mathit{ftv}(\sigma)$ (Exercise 1.3.23). That is, $\exists\sigma$ expresses the requirement that σ have an instance. Type schemes that do not have an instance indicate a type error, so in many situations, one wishes to avoid them; for this reason, we often use the constraint form $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$, which requires σ to have an instance and at the same time associates it with the name $\mathtt{x}$. Because the def form is more primitive, it is easier to work with at a low level, but it is no longer explicitly used after Section 1.3; we always use let instead.

1.3.4 Definition: Environments Γ remain as in Definition 1.2.19, except DM type schemes S are replaced with constrained type schemes σ. We write *dfpi*(Γ) for *dpi*$(\Gamma) \cup$ *fpi*(Γ). We define $\mathsf{def}\ \varnothing\ \mathsf{in}\ C = C$ and $\mathsf{def}\ \Gamma; \mathtt{x} : \sigma\ \mathsf{in}\ C = \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$. Similarly, we define $\mathsf{let}\ \varnothing\ \mathsf{in}\ C = C$ and $\mathsf{let}\ \Gamma; \mathtt{x} : \sigma\ \mathsf{in}\ C = \mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$. We define $\exists\varnothing = \mathsf{true}$ and $\exists(\Gamma; \mathtt{x} : \sigma) = \exists\Gamma \wedge \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\sigma$. □

In order to establish or express certain laws of equivalence between constraints, we need *constraint contexts*. A context is a constraint with zero, one, or several *holes*, written []. The syntax of contexts is as follows:

$$\mathcal{C} ::= [] \mid C \mid \mathcal{C} \wedge \mathcal{C} \mid \exists\bar{\mathtt{X}}.\mathcal{C} \mid \mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C} \mid \mathsf{def}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\mathcal{C}].\mathtt{T}\ \mathsf{in}\ C$$

The application of a constraint context $\mathcal{C}$ to a constraint C, written $\mathcal{C}[C]$, is defined in the usual way. Because a context may have any number of holes, C may disappear or be duplicated in the process. Because a hole may appear in the scope of a binder, some of C's free type variables and free program identifiers may become bound in $\mathcal{C}[C]$. We write *dtv*$(\mathcal{C})$ and *dpi*$(\mathcal{C})$ for the sets of type variables and program identifiers, respectively, that $\mathcal{C}$ may thus capture. We write $\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\mathcal{C}].\mathtt{T}\ \mathsf{in}\ C$ for $\exists\bar{\mathtt{X}}.\mathcal{C} \wedge \mathsf{def}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\mathcal{C}].\mathtt{T}\ \mathsf{in}\ C$. Being able to state such a definition is why we require multi-hole contexts. We let range over *existential constraint contexts*, defined by $\mathcal{X} ::= [] \mid \exists\bar{\mathtt{X}}.\mathcal{X}$.

Meaning

We have defined the syntax of constraints and given an informal description of their meaning. We now give a formal definition of the interpretation of constraints. We begin with the definition of a *model*:

1.3.5 Definition: For every kind κ, let $\mathcal{M}_\kappa$ be a nonempty set, whose elements are the *ground types* of kind κ. In the following, t ranges over $\mathcal{M}_\kappa$, for some κ that may be determined from the context. For every type constructor F of signature $K \Rightarrow \kappa$, let F denote a total function from $\mathcal{M}_K$ into $\mathcal{M}_\kappa$, where the indexed product $\mathcal{M}_K$ is the set of all mappings of domain $dom(K)$ that map every $d \in dom(K)$ to an element of $\mathcal{M}_{K(d)}$. For every predicate P of signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, let P denote a predicate on $\mathcal{M}_{\kappa_1} \times \ldots \times \mathcal{M}_{\kappa_n}$. We require the predicate $\leq$ on $\mathcal{M}_\star \times \mathcal{M}_\star$ to be a partial order. □

For the sake of convenience, we abuse notation and write F for both the type constructor and its interpretation; similarly for predicates. We freely assume that a binary equality predicate, whose interpretation is equality on $\mathcal{M}_\kappa$, is available at every kind κ, so $\mathtt{T}_1 = \mathtt{T}_2$, where $\mathtt{T}_1$ and $\mathtt{T}_2$ have kind κ, is a well-formed constraint.

By varying the set of type constructors, the set of predicates, the set of ground types, and the interpretation of type constructors and predicates, one may define an entire family of related type systems. We informally refer to the collection of these choices as X. Thus, the type systems HM(X) and PCB(X), described in Sections 1.4 and 1.5, are *parameterized* by X.

The following examples give standard ways of defining the set of ground types and the interpretation of type constructors.

1.3.6 Example [Syntactic models]: For every kind κ, let $\mathcal{M}_\kappa$ consist of the *closed* types of kind κ. Then, ground types are types that do not have any free type variables, and form the so-called *Herbrand universe*. Let every type constructor F be interpreted as itself. Models that define ground types and interpret type constructors in this manner are referred to as *syntactic*. □

1.3.7 Example [Tree models]: Let a *path* π be a finite sequence of directions. The empty path is written ϵ and the concatenation of the paths π and π' is written $\pi \cdot \pi'$. Let a *tree* be a partial function t from paths to type constructors whose domain is nonempty and prefix-closed and such that, for every path π in the domain of t, if the type constructor $t(\pi)$ has signature $K \Rightarrow \kappa$, then $\pi \cdot d \in dom(t)$ is equivalent to $d \in dom(K)$ and, furthermore, for every $d \in dom(K)$, the type constructor $t(\pi \cdot d)$ has image kind $K(d)$. If π is in the domain of t, then the *subtree* of t rooted at π, written t/π, is the partial function $\pi' \mapsto t(\pi \cdot \pi')$. A tree is *finite* if and only if it has finite domain. A tree is *regular* if and only if it has a finite number of distinct subtrees. Every finite tree is thus regular. Let $\mathcal{M}_\kappa$ consist of the *finite* (resp. *regular*) trees t such that $t(\epsilon)$ has image kind κ: then, we have a *finite* (resp. *regular*) *tree model*.

If F has signature $K \Rightarrow \kappa$, one may interpret F as the function that maps $T \in \mathcal{M}_K$ to the ground type $t \in \mathcal{M}_\kappa$ defined by $t(\epsilon) = F$ and $t/d = T(d)$ for $d \in dom(T)$, that is, the unique ground type whose head symbol is F and whose subtree rooted at d is $T(d)$. Then, we have a *free* tree model. Please note that free finite tree models coincide with syntactic models, as defined in the previous example. □

Rows (Section 1.11) are interpreted in a tree model, albeit not a free one. The following examples suggest different ways of interpreting the subtyping predicate.

1.3.8 Example [Equality models]: The simplest way of interpreting the subtyping predicate is to let $\leq$ denote equality on every $\mathcal{M}_\kappa$. Models that do so are referred to as *equality models*. When no predicate other than equality is available, we say that the model is *equality-only*. □

1.3.9 EXAMPLE [STRUCTURAL, NONSTRUCTURAL SUBTYPING]: Let a *variance* ν be a nonempty subset of $\{-,+\}$, written $-$ (*contravariant*), $+$ (*covariant*), or $\pm$ (*invariant*) for short. Define the *composition* of two variances as an associative commutative operation with $+$ as neutral element and such that $-- = +$ and $\pm- = \pm\pm = \pm$. Now, consider a free (finite or regular) tree model, where every direction d comes with a fixed variance $\nu(d)$. Define the variance $\nu(\pi)$ of a path π as the composition of the variances of its elements. Let $\leqslant$ be a partial order on type constructors such that (i) if $F_1 \leqslant F_2$ holds and F_1 and F_2 have signature $K_1 \Rightarrow \kappa_1$ and $K_2 \Rightarrow \kappa_2$, respectively, then K_1 and K_2 agree on the intersection of their domains and κ_1 and κ_2 coincide; and (ii) $F_0 \leqslant F_1 \leqslant F_2$ implies $dom(F_0) \cap dom(F_2) \subseteq dom(F_1)$. Let $\leqslant^+$, $\leqslant^-$, and $\leqslant^\pm$ stand for $\leqslant$, $\geqslant$, and $=$, respectively. Then, define the interpretation of subtyping as follows: if $t_1, t_2 \in \mathcal{M}_\kappa$, let $t_1 \leq t_2$ hold if and only if, for every path $\pi \in dom(t_1) \cap dom(t_2)$, $t_1(\pi) \leqslant^{\nu(\pi)} t_2(\pi)$ holds. It is not difficult to check that $\leq$ is a partial order on every $\mathcal{M}_\kappa$. The reader is referred to (Kozen, Palsberg, and Schwartzbach., 1995) for more details about this construction. Models that define subtyping in this manner are referred to as *nonstructural subtyping models*.

A simple nonstructural subtyping model is obtained by letting the directions *domain* and *codomain* be contra- and covariant, respectively, and introducing, in addition to the type constructor $\rightarrow$, two type constructors $\bot$ and $\top$ of signature $\star$. This gives rise to a model where $\bot$ is the least ground type, $\top$ is the greatest ground type, and the arrow type constructor is, as usual, contravariant in its domain and covariant in its codomain.

A typical use of nonstructural subtyping is in type systems for records. One may, for instance, introduce a covariant direction *content* of kind $\star$, a kind $\diamond$, a type constructor abs of signature $\diamond$, a type constructor pre of signature $\{content \mapsto \star\} \Rightarrow \diamond$, and let $\mathsf{pre} \leqslant \mathsf{abs}$. This gives rise to a model where $\mathsf{pre}\ t \leq \mathsf{abs}$ holds for every $t \in \mathcal{M}_\star$. This form of subtyping is called *nonstructural* because comparable ground types may have different shapes, such as $\bot$ and $\bot \rightarrow \top$, or $\mathsf{pre}\ \top$ and abs. Nonstructural subtyping has been studied, for example, in (Kozen, Palsberg, and Schwartzbach., 1995; Palsberg, Wand, and O'Keefe, 1997; Pottier, 2001b; Niehren and Priesnitz, 2003). Section 1.11 says more about typechecking operations on records.

An important particular case arises when *any two type constructors related by $\leqslant$ have the same arity.* In that case, it is not difficult to show that *any two ground types related by subtyping must have the same shape*, that is, if $t_1 \leq t_2$ holds, then $dom(t_1)$ and $dom(t_2)$ coincide. For this reason, such an interpretation of subtyping is usually referred to as *atomic* or *structural* subtyping. It has been studied in the finite (Mitchell, 1984, 1991b; Frey, 1997; Rehof, 1997; Kuncak and Rinard, 2003; Simonet, 2003) and regular (Tiuryn and Wand,

1993) cases. Structural subtyping is often used in automated program analyses that enrich standard types with atomic annotations without altering their shape. □

Our last example suggests a predicate other than equality and subtyping.

1.3.10 Example [Conditional constraints]: Consider a nonstructural subtyping model. For every type constructor F of image kind κ and for every kind κ', let $(F \leqslant \cdot \Rightarrow \cdot \leq \cdot)$ be a predicate of signature $\kappa \otimes \kappa' \otimes \kappa' \Rightarrow \cdot$. Thus, if $\mathtt{T}_0$ has kind κ and $\mathtt{T}_1$, $\mathtt{T}_2$ have the same kind, then $F \leqslant \mathtt{T}_0 \Rightarrow \mathtt{T}_1 \leq \mathtt{T}_2$ is a well-formed constraint, called a *conditional subtyping constraint.* Its interpretation is defined as follows: if $t_0 \in \mathcal{M}_\kappa$ and $t_1, t_2 \in \mathcal{M}_{\kappa'}$, then $F \leqslant t_0 \Rightarrow t_1 \leq t_2$ holds if and only if $F \leqslant t_0(\epsilon)$ implies $t_1 \leq t_2$. In other words, if t_0's head symbol exceeds F according to the ordering on type constructors, then the subtyping constraint $t_1 \leq t_2$ must hold; otherwise, the conditional constraint holds vacuously. Conditional constraints have been studied *e.g.* in (Reynolds, 1969a; Heintze, 1993; Aiken, Wimmers, and Lakshman, 1994; Pottier, 2000; Su and Aiken, 2001). □

Many other kinds of constraints exist; see *e.g.* (Comon, 1993).

Throughout this chapter, we assume (unless stated otherwise) that the set of type constructors, the set of predicates, and the model—which, together, form the parameter X—are arbitrary and fixed.

As usual, the meaning of a constraint is a function of the meaning of its free type variables, which is given by a *ground assignment.* The meaning of free program identifiers may be defined as part of the constraint, if desired, using a def prefix, so it need not be given by a separate assignment.

1.3.11 Definition: A *ground assignment* ϕ is a total, kind-preserving mapping from $\mathcal{V}$ into $\mathcal{M}$. Ground assignments are extended to types by $\phi(F\,\mathtt{T}_1\,\ldots\,\mathtt{T}_n) = F(\phi(\mathtt{T}_1), \ldots, \phi(\mathtt{T}_n))$. Then, for every type $\mathtt{T}$ of kind κ, $\phi(\mathtt{T})$ is a ground type of kind κ. Whether a constraint C holds under a ground assignment ϕ, written $\phi \vdash C$ (read: ϕ *satisfies* C), is defined by the rules in Figure 1-5. A constraint C is *satisfiable* if and only if $\phi \vdash C$ holds for some ϕ. It is *false* if and only if $\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C$ holds for *no* ground assignment ϕ and environment Γ. □

Let us now explain the rules that define constraint satisfaction (Figure 1-5). They are syntax-directed: that is, to a given constraint, at most one rule applies. It is determined by the nature of the first construct that appears under a maximal def prefix. CM-True states that a constraint of the form $\mathsf{def}\ \Gamma\ \mathsf{in\ true}$ is a tautology, that is, holds under every ground assignment. No rule matches constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in\ false}$, which means that such constraints do not have a solution. CM-Predicate states that the meaning

$$\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \mathsf{true} \quad \text{(CM-True)}$$

$$\frac{P(\phi(\mathtt{T}_1), \ldots, \phi(\mathtt{T}_n))}{\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ P\,\mathtt{T}_1 \ldots \mathtt{T}_n} \quad \text{(CM-Predicate)}$$

$$\frac{\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1 \qquad \phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2}{\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2)} \quad \text{(CM-And)}$$

$$\frac{\phi[\vec{\mathtt{X}} \mapsto \vec{t}] \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)}{\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.C} \quad \text{(CM-Exists)}$$

$$\frac{\phi \vdash \mathsf{def}\ \Gamma_1\ \mathsf{in}\ \sigma \preceq \mathtt{T}' \qquad \mathtt{x} \notin \mathit{dpi}(\Gamma_2)}{\phi \vdash \mathsf{def}\ \Gamma_1; \mathtt{x} : \sigma; \Gamma_2\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}'} \quad \text{(CM-Instance)}$$

Figure 1-5: Meaning of constraints

of a predicate application is given by the predicate's interpretation within the model. More specifically, if P's signature is $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \cdot$, then, by well-formedness of the constraint, every $\mathtt{T}_i$ is of kind κ_i, so $\phi(\mathtt{T}_i)$ is a ground type in $\mathcal{M}_{\kappa_i}$. By Definition 1.3.5, P denotes a predicate on $\mathcal{M}_{\kappa_1} \times \ldots \times \mathcal{M}_{\kappa_n}$, so the rule's premise is mathematically well-formed. It is independent of Γ, which is natural, since a predicate application has no free program identifiers. CM-And requires each of the conjuncts to be valid in isolation. The information in Γ is made available to each branch. CM-Exists allows the type variables $\vec{\mathtt{X}}$ to denote arbitrary ground types $\vec{t}$ within C, independently of their image through ϕ. We implicitly require $\vec{\mathtt{X}}$ and $\vec{t}$ to have matching kinds, so that $\phi[\vec{\mathtt{X}} \mapsto \vec{t}]$ remains a kind-preserving ground assignment. The side condition $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)$—which may always be satisfied by suitable α-conversion of the constraint $\exists\bar{\mathtt{X}}.C$—prevents free occurrences of the type variables $\bar{\mathtt{X}}$ within Γ from being unduly affected. CM-Instance concerns constraints of the form $\mathsf{def}\ \Gamma\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}'$. The constraint $\mathtt{x} \preceq \mathtt{T}'$ is turned into $\sigma \preceq \mathtt{T}'$, where, according to the second premise, σ is $\Gamma(\mathtt{x})$. Please recall that constraints of such a form were introduced in Definition 1.3.3. The environment Γ is replaced with a suitable prefix of itself, namely Γ_1, so that the free program identifiers of σ retain their meaning.

It is intuitively clear that the constraints $\mathsf{def}\ \mathtt{x} : \sigma\ \mathsf{in}\ C$ and $[\mathtt{x} \mapsto \sigma]C$ have the same meaning, where the latter denotes the capture-avoiding substitution of σ for $\mathtt{x}$ throughout C. As a matter of fact, it would have been possible to use this equivalence as a *definition* of the meaning of def constraints, but the present style is pleasant as well. This confirms our (informal) claim that the def form is an explicit substitution form.

It is possible for a constraint to be neither satisfiable nor false. Consider, for instance, the constraint $\exists\mathtt{Z}.\mathtt{x} \preceq \mathtt{Z}$. Because the identifier $\mathtt{x}$ is free, CM-Instance is not applicable, so the constraint is not satisfiable. Furthermore, placing it within the context $\mathsf{let}\ \mathtt{x} : \forall \mathtt{X}.\mathtt{X}\ \mathsf{in}\ []$ makes it satisfied by every ground assignment, so it is not false. Here, the assertions "C is satisfiable" and "C is false" are opposite when $fpi(C) = \varnothing$ holds, whereas in a standard first-order logic, they always are.

In a judgement of the form $\phi \vdash C$, the ground assignment ϕ applies to the free type variables of C. This is made precise by the following statements. In the second one, $\circ$ is composition and $\theta(C)$ is the capture-avoiding application of the type substitution θ to C.

1.3.12 LEMMA: If $\bar{\mathtt{X}} \# ftv(C)$ holds, then $\phi \vdash C$ and $\phi[\vec{\mathtt{X}} \mapsto \vec{t}] \vdash C$ are equivalent. $\square$

1.3.13 LEMMA: $\phi \circ \theta \vdash C$ and $\phi \vdash \theta(C)$ are equivalent. $\square$

Reasoning with constraints

Because constraints lie at the heart of our treatment of ML-the-type-system, most of our proofs involve establishing logical properties of constraints, that is, *entailment* or *equivalence* assertions. Let us first define these notions.

1.3.14 DEFINITION: We write $C_1 \Vdash C_2$, and say that C_1 *entails* C_2, if and only if, for every ground assignment ϕ and for every environment Γ, $\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C_1$ implies $\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C_2$. We write $C_1 \equiv C_2$, and say that C_1 and C_2 are *equivalent*, if and only if $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$ hold. $\square$

This definition measures the strength of a constraint by the set of pairs (ϕ, Γ) that satisfy it, and considers a constraint stronger if fewer such pairs satisfy it. In other words, C_1 entails C_2 when C_1 imposes stricter requirements on its free type variables and program identifiers than C_2 does. We remark that C is false if and only if $C \equiv \mathsf{false}$ holds. It is straightforward to check that entailment is reflexive and transitive and that $\equiv$ is indeed an equivalence relation.

We immediately exploit the notion of constraint equivalence to define what it means for a type constructor to be covariant, contravariant, or invariant with respect to one of its parameters. Let F be a type constructor of signature $\kappa_1 \otimes \ldots \otimes \kappa_n \Rightarrow \kappa$. Let $i \in \{1, \ldots, n\}$. F is *covariant* (resp. *contravariant*, *invariant*) with respect to its i^{th} parameter if and only if, for all types $\mathtt{T}_1, \ldots, \mathtt{T}_n$ and $\mathtt{T}'_i$ of appropriate kinds, the constraint $F\,\mathtt{T}_1 \ldots \mathtt{T}_i \ldots \mathtt{T}_n \leq F\,\mathtt{T}_1 \ldots \mathtt{T}'_i \ldots \mathtt{T}_n$ is equivalent to $\mathtt{T}_i \leq \mathtt{T}'_i$ (resp. $\mathtt{T}'_i \leq \mathtt{T}_i$, $\mathtt{T}_i = \mathtt{T}'_i$). We let the reader check the following facts: (i) in an equality model, these three notions coincide; (ii) in an equality free tree model, every type constructor is invariant with respect to each of its parameters; and (iii) in a nonstructural subtyping model, if the direction d has been declared covariant (resp. contravariant, invariant), then every type constructor whose arity includes d is covariant (resp. contravariant, invariant) with respect to d. In the following, *we require the type constructor $\rightarrow$ to be contravariant with respect to its domain and covariant with respect to its codomain*—a standard requirement in type systems with subtyping (TAPL Chapter 15). These properties are summed up by the following equivalence law:

$$\mathtt{T}_1 \rightarrow \mathtt{T}_2 \leq \mathtt{T}'_1 \rightarrow \mathtt{T}'_2 \equiv \mathtt{T}'_1 \leq \mathtt{T}_1 \wedge \mathtt{T}_2 \leq \mathtt{T}'_2 \qquad \text{(C-Arrow)}$$

Please note that this is a high-level requirement about the interpretation of types and of the subtyping predicate. In an equality free tree model, for instance, it is always satisfied. In a nonstructural subtyping model, it boils down to requiring that the directions *domain* and *codomain* be declared contravariant and covariant, respectively. In the general case, we do not have any knowledge of the model, and cannot formulate a more precise requirement. Thus, it is up to the designer of the model to ensure that C-Arrow holds.

We also exploit the notion of constraint equivalence to define what it means for two type constructors to be incompatible. Two type constructors F_1 and F_2 with the same image kind are *incompatible* if and only if all constraints of the form $F_1\,\vec{\mathtt{T}}_1 \leq F_2\,\vec{\mathtt{T}}_2$ and $F_2\,\vec{\mathtt{T}}_2 \leq F_1\,\vec{\mathtt{T}}_1$ are false; then, we write $F_1 \bowtie F_2$. Please note that in an equality free tree model, any two distinct type constructors are incompatible. In the following, we often indicate that a newly introduced type constructor must be *isolated*. We implicitly require that, whenever each of F_1 and F_2 is isolated, F_1 and F_2 be incompatible. Thus, the notion of "isolation" provides a concise and modular way of stating a collection of incompatibility requirements. We consider the type constructor $\rightarrow$ isolated.

Entailment is preserved by arbitrary constraint contexts, as stated by the following theorem. As a result, constraint equivalence is a congruence. Throughout this chapter, these facts are often used implicitly.

1.3.15 Theorem [Congruence]: $C_1 \Vdash C_2$ implies $\mathcal{C}[C_1] \Vdash \mathcal{C}[C_2]$. □

We now give a series of lemmas that provide useful entailment laws.

The following is a standard property of existential quantification.

1.3.16 Lemma: $C \Vdash \exists\bar{\mathtt{X}}.C$. □

The following lemma states that any supertype of an instance of σ is also an instance of σ.

1.3.17 Lemma: $\sigma \preceq \mathtt{T} \wedge \mathtt{T} \leq \mathtt{T}' \Vdash \sigma \preceq \mathtt{T}'$. □

The next lemma gives another interesting simplification law.

1.3.18 Lemma: $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ implies $\exists\mathtt{X}.(\mathtt{X} = \mathtt{T}) \equiv \mathsf{true}$. □

The following lemma states that, provided D is satisfied, the type $\mathtt{T}$ is an instance of the constrained type scheme $\forall\bar{\mathtt{X}}[D].\mathtt{T}$.

1.3.19 LEMMA: $D \Vdash \forall\bar{\mathtt{X}}[D].\mathtt{T} \preceq \mathtt{T}$. □

This technical lemma helps justify Definition 1.3.21 below.

1.3.20 LEMMA: Let $\mathtt{Z} \notin \mathit{ftv}(C, \sigma, \mathtt{T})$. Then, $C \Vdash \sigma \preceq \mathtt{T}$ holds if and only if $C \wedge \mathtt{T} \leq \mathtt{Z} \Vdash \sigma \preceq \mathtt{Z}$ holds. □

It is useful to define what it means for a type scheme σ_1 to be *more general* than a type scheme σ_2. Our informal intent is for $\sigma_1 \preceq \sigma_2$ to mean: *every instance of σ_2 is an instance of σ_1*. In Definition 1.3.3, we have introduced the constraint form $\sigma \preceq \mathtt{T}$ as syntactic sugar. Similarly, one might wish to make $\sigma_1 \preceq \sigma_2$ a derived constraint form; however, this is impossible, because neither universal quantification nor implication are available in the constraint language. We can, however, exploit the fact that these logical connectives are implicit in entailment assertions by defining a judgement of the form $C \Vdash \sigma_1 \preceq \sigma_2$, whose meaning is: *under the constraint C, σ_1 is more general than σ_2.*

1.3.21 DEFINITION: We write $C \Vdash \sigma_1 \preceq \sigma_2$ if and only if $\mathtt{Z} \notin \mathit{ftv}(C, \sigma_1, \sigma_2)$ implies $C \wedge \sigma_2 \preceq \mathtt{Z} \Vdash \sigma_1 \preceq \mathtt{Z}$. We write $C \Vdash \sigma_1 \equiv \sigma_2$ when both $C \Vdash \sigma_1 \preceq \sigma_2$ and $C \Vdash \sigma_2 \preceq \sigma_1$ hold. □

This notation is not ambiguous because the assertion $C \Vdash \sigma \preceq \mathtt{T}$, whose meaning was initially given by Definitions 1.3.3 and 1.3.14, retains the same meaning under the new definition—this is shown by Lemma 1.3.20 above.

The next lemma provides a way of exploiting the ordering between type schemes introduced by Definition 1.3.21. It states that a type scheme occurs in *contravariant* position when it is within a def prefix. In other words, the more general the type scheme, the weaker the entire constraint.

1.3.22 LEMMA: $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \mathsf{def}\ \mathtt{x} : \sigma_2\ \mathsf{in}\ D \Vdash \mathsf{def}\ \mathtt{x} : \sigma_1\ \mathsf{in}\ D$. □

The following exercise generalizes this result to let forms.

1.3.23 EXERCISE [★★, $\not\rightarrow$]: Prove that $\mathtt{Z} \notin \mathit{ftv}(\sigma)$ implies $\exists\sigma \equiv \exists\mathtt{Z}.\sigma \preceq \mathtt{Z}$. Explain why, as a result, $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \exists\sigma_2 \Vdash \exists\sigma_1$. Use this fact to prove that $C \Vdash \sigma_1 \preceq \sigma_2$ implies $C \wedge \mathsf{let}\ \mathtt{x} : \sigma_2\ \mathsf{in}\ D \Vdash \mathsf{let}\ \mathtt{x} : \sigma_1\ \mathsf{in}\ D$. □

The next lemma states that, modulo equivalence, the only constraint that constrains x without explicitly referring to it is false.

1.3.24 LEMMA: $C \Vdash \mathtt{x} \preceq \mathtt{T}$ and $\mathtt{x} \notin \mathit{fpi}(C)$ imply $C \equiv \mathsf{false}$. □

The following lemma states that the more universal quantifiers are present, the more general the type scheme.

1.3.25 LEMMA: $\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2 \Vdash \mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2$. □

Conversely, and perhaps surprisingly, it is sometimes possible to *remove* some type variables from the universal quantifier prefix of a type scheme without compromising its generality. This is the case when the value of these type variables is *determined* in a unique way. In short, C determines $\bar{\mathtt{Y}}$ if and only if, given the values of $ftv(C) \setminus \bar{\mathtt{Y}}$ and given that C holds, it is possible to reconstruct, in a unique way, the values of $\bar{\mathtt{Y}}$.

1.3.26 DEFINITION: *C determines* $\bar{\mathtt{Y}}$ if and only if, for every environment Γ, two ground assignments that satisfy $\mathsf{def}\ \Gamma\ \mathsf{in}\ C$ and that coincide outside $\bar{\mathtt{Y}}$ must coincide on $\bar{\mathtt{Y}}$ as well. □

Two concrete instances of determinacy, one of which is valid only in free tree models, are given by Lemma 1.8.7 on page 82. Determinacy is exploited by the equivalence law C-LETALL in Figure 1-6.

We now give a toolbox of constraint equivalence laws. It is worth noting that they do not form a complete axiomatization of constraint equivalence—in fact, they cannot, since the syntax and meaning of constraints is partly unspecified.

1.3.27 THEOREM: All equivalence laws in Figure 1-6 hold. □

Let us explain. C-AND and C-ANDAND state that conjunction is commutative and associative. C-DUP states that redundant conjuncts may be freely added or removed, where a conjunct is redundant if and only if it is entailed by another conjunct. Throughout this chapter, these three laws are often used implicitly. C-EXEX and C-EX* allow grouping consecutive existential quantifiers and suppressing redundant ones, where a quantifier is redundant if and only if it does not occur free within its scope. C-EXAND allows conjunction and existential quantification to commute, provided no capture occurs; it is known as a *scope extrusion* law. When the rule is oriented from left to right, its side-condition may always be satisfied by suitable α-conversion. C-EXTRANS states that it is equivalent for a type $\mathtt{T}'$ to be an instance of σ or to be a supertype of some instance of σ. We remark that the instances of a monotype are its supertypes, that is, by Definition 1.3.3, $\mathtt{T} \preceq \mathtt{T}'$ and $\mathtt{T} \leq \mathtt{T}'$ are equivalent. As a result, specializing C-EXTRANS to the case where σ is a monotype, we find that $\mathtt{T} \leq \mathtt{T}'$ is equivalent to $\exists\mathtt{Z}.(\mathtt{T} \leq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}')$, for fresh $\mathtt{Z}$, a standard equivalence law. When oriented from left to right, it becomes an interesting *simplification* law: in a chain of subtyping constraints, an intermediate variable such as $\mathtt{Z}$ may be suppressed, provided it is *local*, as witnessed by the existential quantifier $\exists\mathtt{Z}$. C-INID states that, within the scope of the binding $\mathtt{x} : \sigma$, every *free* occurrence of $\mathtt{x}$ may be safely replaced with σ. The restriction to free occurrences stems from the side-condition $\mathtt{x} \notin dpi(\mathcal{C})$. When the

$$C_1 \wedge C_2 \equiv C_2 \wedge C_1 \qquad \text{(C-\textsc{And})}$$

$$(C_1 \wedge C_2) \wedge C_3 \equiv C_1 \wedge (C_2 \wedge C_3) \qquad \text{(C-\textsc{AndAnd})}$$

$$C_1 \wedge C_2 \equiv C_1 \qquad \text{if } C_1 \Vdash C_2 \qquad \text{(C-\textsc{Dup})}$$

$$\exists\bar{\mathtt{X}}.\exists\bar{\mathtt{Y}}.C \equiv \exists\bar{\mathtt{X}}\bar{\mathtt{Y}}.C \qquad \text{(C-\textsc{ExEx})}$$

$$\exists\bar{\mathtt{X}}.C \equiv C \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(C) \qquad \text{(C-\textsc{Ex}*)}$$

$$(\exists\bar{\mathtt{X}}.C_1) \wedge C_2 \equiv \exists\bar{\mathtt{X}}.(C_1 \wedge C_2) \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(C_2) \qquad \text{(C-\textsc{ExAnd})}$$

$$\exists\mathtt{Z}.(\sigma \preceq \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T}') \equiv \sigma \preceq \mathtt{T}' \qquad \text{if } \mathtt{Z} \notin \mathit{ftv}(\sigma, \mathtt{T}') \qquad \text{(C-\textsc{ExTrans})}$$

$$\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}[\mathtt{x} \preceq \mathtt{T}'] \equiv \mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}[\sigma \preceq \mathtt{T}'] \qquad \text{(C-\textsc{InId})}$$
$$\text{if } \mathtt{x} \notin \mathit{dpi}(\mathcal{C}) \text{ and } \mathit{dtv}(\mathcal{C}) \# \mathit{ftv}(\sigma) \text{ and } \{\mathtt{x}\} \cup \mathit{dpi}(\mathcal{C}) \# \mathit{fpi}(\sigma)$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ C \equiv \exists\Gamma \wedge C \qquad \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C) \qquad \text{(C-\textsc{In}*)}$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2) \equiv (\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1) \wedge (\mathsf{let}\ \Gamma\ \mathsf{in}\ C_2) \qquad \text{(C-\textsc{InAnd})}$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2) \equiv (\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1) \wedge C_2 \qquad \text{if } \mathit{dpi}(\Gamma) \# \mathit{fpi}(C_2) \qquad \text{(C-\textsc{InAnd}*)}$$

$$\mathsf{let}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.C \equiv \exists\bar{\mathtt{X}}.\mathsf{let}\ \Gamma\ \mathsf{in}\ C \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma) \qquad \text{(C-\textsc{InEx})}$$

$$\mathsf{let}\ \Gamma_1; \Gamma_2\ \mathsf{in}\ C \equiv \mathsf{let}\ \Gamma_2; \Gamma_1\ \mathsf{in}\ C \qquad \text{(C-\textsc{LetLet})}$$
$$\text{if } \mathit{dpi}(\Gamma_1) \# \mathit{dpi}(\Gamma_2) \text{ and } \mathit{dpi}(\Gamma_2) \# \mathit{fpi}(\Gamma_1) \text{ and } \mathit{dpi}(\Gamma_1) \# \mathit{fpi}(\Gamma_2)$$

$$\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[C_1 \wedge C_2].\mathtt{T}\ \mathsf{in}\ C_3 \equiv C_1 \wedge \mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[C_2].\mathtt{T}\ \mathsf{in}\ C_3 \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(C_1) \qquad \text{(C-\textsc{LetAnd})}$$

$$\mathsf{let}\ \Gamma; \mathtt{x} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2 \equiv \mathsf{let}\ \Gamma; \mathtt{x} : \forall\bar{\mathtt{X}}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1].\mathtt{T}\ \mathsf{in}\ C_2 \qquad \text{(C-\textsc{LetDup})}$$
$$\text{if } \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma) \text{ and } \mathit{dpi}(\Gamma) \# \mathit{fpi}(\Gamma)$$

$$\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\exists\bar{\mathtt{Y}}.C_1].\mathtt{T}\ \mathsf{in}\ C_2 \equiv \mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2 \qquad \text{if } \bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}) \qquad \text{(C-\textsc{LetEx})}$$

$$\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2 \equiv \exists\bar{\mathtt{Y}}.\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2 \qquad \text{(C-\textsc{LetAll})}$$
$$\text{if } \bar{\mathtt{Y}} \# \mathit{ftv}(C_2) \text{ and } \exists\bar{\mathtt{X}}.C_1 \text{ determines } \bar{\mathtt{Y}}$$

$$\exists\mathtt{X}.(\mathtt{T} \leq \mathtt{X} \wedge \mathsf{let}\ \mathtt{x} : \mathtt{X}\ \mathsf{in}\ C) \equiv \mathsf{let}\ \mathtt{x} : \mathtt{T}\ \mathsf{in}\ C \qquad \text{if } \mathtt{X} \notin \mathit{ftv}(\mathtt{T}, C) \qquad \text{(C-\textsc{LetSub})}$$

$$\vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]C \equiv \vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge C \qquad \text{(C-\textsc{Eq})}$$

$$\mathsf{true} \equiv \exists\bar{\mathtt{X}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}}) \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}}) \qquad \text{(C-\textsc{Name})}$$

$$[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]C \equiv \exists\bar{\mathtt{X}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge C) \qquad \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}}) \qquad \text{(C-\textsc{NameEq})}$$

Figure 1-6: Constraint equivalence laws

rule is oriented from left to right, its other side-conditions, which require the context $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \mathcal{C}$ not to capture σ's free type variables or free program identifiers, may always be satisfied by suitable α-conversion. C-In* complements the previous rule by allowing redundant let bindings to be simplified. We remark that C-InId and C-In* provide a simple procedure for eliminating let forms. C-InAnd states that the let form commutes with conjunction; C-InAnd* spells out a common particular case. C-InEx states that it commutes with existential quantification. When the rule is oriented from left to right, its side-condition may always be satisfied by suitable α-conversion. C-LetLet states that let forms may commute, provided they bind distinct program identifiers and provided no free program identifiers are captured in the process. C-LetAnd allows the conjunct C_1 to be moved outside of the constrained type scheme $\forall\bar{\mathtt{X}}[C_1 \wedge C_2].\mathtt{T}$, provided it does not involve any of the universally quantified type variables $\bar{\mathtt{X}}$. When oriented from left to right, the rule yields an important simplification law: indeed, taking an instance of $\forall\bar{\mathtt{X}}[C_2].\mathtt{T}$ is less expensive than taking an instance of $\forall\bar{\mathtt{X}}[C_1 \wedge C_2].\mathtt{T}$, since the latter involves creating a copy of C_1, while the former does not. C-LetDup allows pushing a series of let bindings into a constrained type scheme, provided no capture occurs in the process. It is not used as a simplification law but as a tool in some proofs. C-LetEx states that it does not make any difference for a set of type variables $\bar{\mathtt{Y}}$ to be existentially quantified inside a constrained type scheme or part of the type scheme's universal quantifiers. Indeed, in either case, taking an instance of the type scheme means producing a constraint where $\bar{\mathtt{Y}}$ is existentially quantified. C-LetAll provides a restricted converse of Lemma 1.3.25. Together, C-LetEx and C-LetAll allow—in some situations only—to hoist existential quantifiers out of the *left*-hand side of a let form.

1.3.28 Example: C-LetAll would be invalid without the condition that $\exists\bar{\mathtt{X}}.C_1$ determines $\bar{\mathtt{Y}}$. Consider, for instance, the constraint $\mathsf{let}\ \mathtt{x} : \forall\mathtt{Y}.\mathtt{Y} \rightarrow \mathtt{Y}\ \mathsf{in}\ (\mathtt{x} \preceq \mathtt{int} \rightarrow \mathtt{int} \wedge \mathtt{x} \preceq \mathtt{bool} \rightarrow \mathtt{bool})$ **(1)**, where int and bool are incompatible nullary type constructors. By C-InId and C-In*, it is equivalent to $\exists\mathtt{Y}.(\mathtt{Y} \rightarrow \mathtt{Y} \leq \mathtt{int} \rightarrow \mathtt{int}) \wedge \exists\mathtt{Y}.(\mathtt{Y} \rightarrow \mathtt{Y} \leq \mathtt{bool} \rightarrow \mathtt{bool})$, that is, true. Now, if C-LetAll was valid without its side-condition, then (1) would also be equivalent to $\exists\mathtt{Y}.\mathsf{let}\ \mathtt{x} : \mathtt{Y} \rightarrow \mathtt{Y}\ \mathsf{in}\ (\mathtt{x} \preceq \mathtt{int} \rightarrow \mathtt{int} \wedge \mathtt{x} \preceq \mathtt{bool} \rightarrow \mathtt{bool})$, which by C-InId and C-In* is $\exists\mathtt{Y}.(\mathtt{Y} \rightarrow \mathtt{Y} \leq \mathtt{int} \rightarrow \mathtt{int} \wedge \mathtt{Y} \rightarrow \mathtt{Y} \leq \mathtt{bool} \rightarrow \mathtt{bool})$. By C-Arrow and C-ExTrans, this is $\mathtt{int} = \mathtt{bool}$, that is, false. Thus, the law is invalid in this case. It is easy to see why: when the type scheme σ contains a $\forall\mathtt{Y}$ quantifier, every instance of σ receives its own $\exists\mathtt{Y}$ quantifier, making $\mathtt{Y}$ a distinct (local) type variable; when $\mathtt{Y}$ is not universally quantified, however, all instances of σ *share* references to a single (global) type variable $\mathtt{Y}$. This corresponds to the intuition that, in the former case, σ is *polymorphic* in $\mathtt{Y}$, while in the latter case, it is *monomorphic* in $\mathtt{Y}$. Lemma 1.3.25 states that, when deprived of its side-condition, C-LetAll is only an entailment law, as opposed to an equivalence law. Similarly, it is in general invalid to hoist an existential quantifier out of the left-hand side of a let form. To see this, one may study the (equivalent) constraint $\mathsf{let}\ \mathtt{x} : \forall\mathtt{X}[\exists\mathtt{Y}.\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Y}].\mathtt{X}\ \mathsf{in}\ (\mathtt{x} \preceq \mathtt{int} \rightarrow \mathtt{int} \wedge \mathtt{x} \preceq \mathtt{bool} \rightarrow \mathtt{bool})$.

Naturally, in the above examples, the side-condition "$\mathtt{true}$ determines $\mathtt{Y}$" does *not* hold: by Definition 1.3.26, it is equivalent to "two ground assignments that coincide outside $\mathtt{Y}$ must coincide on $\mathtt{Y}$ as well", which is false as soon as $\mathcal{M}_\star$ contains two distinct elements, such as $\mathtt{int}$ and $\mathtt{bool}$ here. There are cases, however, where the side-condition does hold. For instance, we later prove that $\exists\mathtt{X}.\mathtt{Y} = \mathtt{int}$ determines $\mathtt{Y}$; see Lemma 1.8.7. As a result, C-LetAll states that $\mathsf{let}\ \mathtt{x} : \forall\mathtt{XY}[\mathtt{Y} = \mathtt{int}].\mathtt{Y} \rightarrow \mathtt{X}\ \mathsf{in}\ C$ **(1)** is equivalent to $\exists\mathtt{Y}.\mathsf{let}\ \mathtt{x} : \forall\mathtt{X}[\mathtt{Y} = \mathtt{int}].\mathtt{Y} \rightarrow \mathtt{X}\ \mathsf{in}\ C$ **(2)**, provided $\mathtt{Y} \notin \mathit{ftv}(C)$. The intuition is simple: because $\mathtt{Y}$ is forced to assume the value $\mathtt{int}$ by the equation $\mathtt{Y} = \mathtt{int}$, it makes no difference whether $\mathtt{Y}$ is or isn't universally quantified. We remark that, by C-LetAnd, (2) is equivalent to $\exists\mathtt{Y}.(\mathtt{Y} = \mathtt{int} \wedge \mathsf{let}\ \mathtt{x} : \forall\mathtt{X}.\mathtt{Y} \rightarrow \mathtt{X}\ \mathsf{in}\ C)$ **(3)**. In an efficient constraint solver, simplifying (1) into (3) *before* using C-InId to eliminate the let form is worthwhile, since doing so obviates the need for copying the type variable $\mathtt{Y}$ and the equation $\mathtt{Y} = \mathtt{int}$ at every free occurrence of $\mathtt{x}$ inside C. □

C-LetSub is the analogue of an environment strengthening lemma: roughly speaking, it states that, if a constraint holds under the assumption that $\mathtt{x}$ has type $\mathtt{X}$, where $\mathtt{X}$ is some supertype of $\mathtt{T}$, then it also holds under the assumption that $\mathtt{x}$ has type $\mathtt{T}$. The last three rules deal with the equality predicate. C-Eq states that it is valid to replace equals with equals; note the absence of a side-condition. When oriented from left to right, C-Name allows introducing fresh names $\vec{\mathtt{X}}$ for the types $\vec{\mathtt{T}}$. As always, $\vec{\mathtt{X}}$ stands for a vector of *distinct* type variables. Of course, this makes sense only if the definition is not circular, that is, if the type variables $\bar{\mathtt{X}}$ do not occur free within the terms $\bar{\mathtt{T}}$. When oriented from right to left, C-Name may be viewed as a simplification law: it allows eliminating type variables whose value has been determined. C-NameEq is a combination of C-Eq and C-Name. It shows that applying an idempotent substitution to a constraint C amounts to placing C within a certain context. This immediately yields a proof of the following fact:

1.3.29 Lemma: $C \Vdash D$ implies $\theta(C) \Vdash \theta(D)$. □

It is important to stress that, because the effect of a type substitution may be emulated using equations, conjunction, and existential quantification, there is no need ever to employ type substitutions in the definition of a constraint-based type system—it is possible, instead, to express every concept in terms of constraints. In this chapter, we follow this route, and use type substitutions only when dealing with the type system DM, whose historical formulation is substitution-based.

So far, we have considered def a primitive constraint form and defined the let form in terms of def, conjunction, and existential quantification. The motivation for this approach was to simplify the proof of several constraint equivalence laws. However, in the remainder of this chapter, we work with let forms exclusively and never employ the def construct. As a result, *it is possible, from here on, to discard* def *and pretend that* let *is primitive.* This change in perspective offers us a few extra properties, stated in the next two lemmas. First, every constraint that contains a false subconstraint must be false. Second, no satisfiable constraint has a free program identifier.

1.3.30 LEMMA: $\mathcal{C}[\mathsf{false}] \equiv \mathsf{false}$. $\square$

1.3.31 LEMMA: If C is satisfiable, then $\mathit{fpi}(C) = \varnothing$. $\square$

Reasoning with constraints in an equality-only syntactic model

We have given a number of equivalence laws that are valid with respect to any interpretation of constraints, that is, within any model. However, an important special case is that of *equality-only syntactic models*. Indeed, in that specific setting, our constraint-based type systems are in close correspondence with DM. In short, we aim to prove that every satisfiable constraint admits a *canonical solved form*, to show that this notion corresponds to the standard concept of a *most general unifier*, and to establish a few technical properties of most general unifiers.

Thus, let us now assume that constraints are interpreted in an equality-only syntactic model. Let us further assume that, for every kind κ, (i) there are at least *two* type constructors of image kind κ and (ii) for every type constructor F of image kind κ, there exists $t \in \mathcal{M}_\kappa$ such that $t(\epsilon) = F$. We refer to models that violate (i) or (ii) as *degenerate*; one may argue that such models are of little interest. The assumption that the model is nondegenerate is used in the proof of Lemmas 1.3.32 and 1.3.39.

Under these new assumptions, the interpretation of equality coincides with its syntax: every equation that holds in the model is in fact a syntactic truism. The converse, of course, holds in every model.

1.3.32 LEMMA: If $\mathsf{true} \Vdash \mathtt{T} = \mathtt{T}'$ holds, then $\mathtt{T}$ and $\mathtt{T}'$ coincide. $\square$

In a syntactic model, ground types are finite trees. As a result, cyclic equations, such as $\mathtt{X} = \mathtt{int} \rightarrow \mathtt{X}$, are false.

1.3.33 LEMMA: $\mathtt{X} \in \mathit{ftv}(\mathtt{T})$ and $\mathtt{T} \notin \mathcal{V}$ imply $(\mathtt{X} = \mathtt{T}) \equiv \mathsf{false}$. $\square$

A *solved form* is a conjunction of equations, where the left-hand sides are *distinct* type variables that do not appear in the right-hand sides, possibly surrounded by a number of existential quantifiers. Our definition is identical to Lassez, Maher and Marriott's solved forms (1988) and to Jouannaud and Kirchner's *tree* solved forms (1991), except we allow for prenex existential quantifiers, which are made necessary by our richer constraint language. Jouannaud and Kirchner also define *dag* solved forms, which may be exponentially smaller. Because we define solved forms only for proof purposes, we need not take performance into account at this point. The efficient constraint solver presented in Section 1.8 does manipulate graphs, rather than trees. Type scheme introduction and instantiation constructs cannot appear within solved forms; indeed, provided the constraint at hand has no free program identifiers, they can be expanded away. For this reason, their presence in the constraint language has no impact on the results contained in this section.

1.3.34 Definition: A *solved form* is of the form $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}})$. □

Solved forms offer a convenient way of reasoning about constraints because every satisfiable constraint is equivalent to one. In other words, every constraint is equivalent to either a solved form or false. This property is established by the following lemma, whose proof provides a simple but effective procedure to rewrite a constraint to either a solved form or false.

1.3.35 Lemma: Let $\mathit{fpi}(C) = \varnothing$. Then, C is equivalent to either a solved form or false. □

Proof: We first establish that every conjunction of equations is equivalent to either a solved form or false. To do so, we present Robinson's unification algorithm (1971) as a rewriting system. The system's invariant is to operate on constraints of the form either $\vec{\mathtt{X}} = \vec{\mathtt{T}}; C$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}}, C)$ and the semicolon is interpreted as a distinguished conjunction, or false. We identify equations in C up to commutativity. The system is defined as follows:

$$\begin{array}{lrcl}
\vec{\mathtt{X}} = \vec{\mathtt{T}}; & \mathtt{X} = \mathtt{X} \wedge C & \rightarrow & \vec{\mathtt{X}} = \vec{\mathtt{T}}; C \\
\vec{\mathtt{X}} = \vec{\mathtt{T}}; & F\,\vec{\mathtt{T}}_1 = F\,\vec{\mathtt{T}}_2 \wedge C & \rightarrow & \vec{\mathtt{X}} = \vec{\mathtt{T}}; \vec{\mathtt{T}}_1 = \vec{\mathtt{T}}_2 \wedge C \\
\vec{\mathtt{X}} = \vec{\mathtt{T}}; & F_1\,\vec{\mathtt{T}}_1 = F_2\,\vec{\mathtt{T}}_2 \wedge C & \rightarrow & \mathsf{false} \\
 & & & \text{if } F_1 \neq F_2 \\
\vec{\mathtt{X}} = \vec{\mathtt{T}}; & \mathtt{X} = \mathtt{T} \wedge C & \rightarrow & \vec{\mathtt{X}} = [\mathtt{X} \mapsto \mathtt{T}]\vec{\mathtt{T}} \wedge \mathtt{X} = \mathtt{T}; [\mathtt{X} \mapsto \mathtt{T}]C \\
 & & & \text{if } \mathtt{X} \notin \mathit{ftv}(\mathtt{T}) \\
\vec{\mathtt{X}} = \vec{\mathtt{T}}; & \mathtt{X} = \mathtt{T} \wedge C & \rightarrow & \mathsf{false} \\
 & & & \text{if } \mathtt{X} \in \mathit{ftv}(\mathtt{T}) \text{ and } \mathtt{T} \notin \mathcal{V}
\end{array}$$

It is straightforward to check that the above invariant is indeed preserved by the rewriting system. Let us check that constraint equivalence is also preserved. For the first rule, this is immediate. For the second and third rules, it follows from the fact that we have assumed a free tree model; for the fourth rule, a consequence of C-EQ; for the last rule, a consequence of Lemma 1.3.33. Furthermore, the system is terminating; this is witnessed by an ordering where false is the least element and where constraints of the form $\vec{\mathtt{X}} = \vec{\mathtt{T}}; C$ are ordered lexicographically, first by the number of type variables that appear free within C, second by the size of C. Last, a normal form for this rewriting system must be of the form either $\vec{\mathtt{X}} = \vec{\mathtt{T}}; \mathsf{true}$, where (by the invariant) $\bar{\mathtt{X}} \# \mathit{ftv}(\bar{\mathtt{T}})$—that is, a solved form, or false.

Next, we show that the present lemma holds when C is built out of equations, conjunction, and existential quantification. Orienting C-EXAND from left to right yields a terminating rewriting system that preserves constraint equivalence. The normal form of C must be $\exists\bar{\mathtt{Y}}.C'$, where C' is a conjunction of equations. By the previous result, C' is equivalent to either a solved form or false. Because solved forms are preserved by existential quantification and because $\exists\bar{\mathtt{Y}}.\mathsf{false}$ is false, the same holds of C.

Last, we establish the result in the general case. We assume $\mathit{fpi}(C) = \varnothing$ **(1)**. Orienting C-INID and C-IN* from left to right yields a terminating rewriting system that preserves constraint equivalence. The normal form C' of C cannot contain any type scheme introduction forms; given (1), it cannot contain any instantiation forms either. Thus, C' is built out of equations, conjunction, and existential quantification only. By the previous result, it is equivalent to either a solved form or false, which implies that the same holds of C. □

It is possible to impose further restrictions on solved forms. A solved form $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$ is *canonical* if and only if its free type variables are exactly $\bar{\mathtt{X}}$. This is stated, in an equivalent way, by the following definition.

1.3.36 DEFINITION: A *canonical solved form* is a constraint of the form $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, where $\mathit{ftv}(\bar{\mathtt{T}}) \subseteq \bar{\mathtt{Y}}$ and $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$. □

1.3.37 LEMMA: Every solved form is equivalent to a canonical solved form. □

It is easy to describe the solutions of a canonical solved form: they are the ground refinements of the substitution $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$.

1.3.38 LEMMA: A ground assignment ϕ satisfies a canonical solved form $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$ if and only if there exists a ground assignment ϕ' such that $\phi(\vec{\mathtt{X}}) = \phi'(\vec{\mathtt{T}})$. As a result, every canonical solved form is satisfiable. □

Proof: Let $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$ be a canonical solved form. By CM-EXISTS and CM-PREDICATE, ϕ satisfies $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$ if and only if there exists $\vec{t}$ such that $\phi[\vec{\mathtt{Y}} \mapsto \vec{t}](\vec{\mathtt{X}}) = \phi[\vec{\mathtt{Y}} \mapsto \vec{t}](\vec{\mathtt{T}})$. Thanks to the hypotheses $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$ and $\mathit{ftv}(\bar{\mathtt{T}}) \subseteq \bar{\mathtt{Y}}$, this is equivalent to the existence of a ground assignment ϕ' such that $\phi(\vec{\mathtt{X}}) = \phi'(\vec{\mathtt{T}})$.

Thus, for every ground assignment ϕ', $\phi'[\vec{\mathtt{X}} \mapsto \phi'(\vec{\mathtt{T}})]$ satisfies $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, which proves that this constraint is satisfiable. □

Together, Lemmas 1.3.37 and 1.3.38 imply that every solved form is satisfiable. Our interest in canonical solved forms stems from the following fundamental property, which provides a *syntactic* characterization of entailment between canonical solved forms: if $\exists\bar{\mathtt{Y}}_1.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_1)$ is more specific than $\exists\bar{\mathtt{Y}}_2.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_2)$, in a logical sense, then $\vec{\mathtt{T}}_1$ refines $\vec{\mathtt{T}}_2$, in a syntactic sense. The converse also holds (can you prove it?), but is not needed here.

1.3.39 Lemma: If $\exists\bar{\mathtt{Y}}_1.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_1) \Vdash \exists\bar{\mathtt{Y}}_2.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_2)$, where both sides are canonical solved forms, then there exists a type substitution φ such that $\vec{\mathtt{T}}_1 = \varphi(\vec{\mathtt{T}}_2)$. □

As a corollary, we find that canonical solved forms are *unique* up to α-conversion and up to C-Ex*, *provided* the set $\bar{\mathtt{X}}$ of their free type variables is fixed.

1.3.40 Lemma: If the canonical solved forms $\exists\bar{\mathtt{Y}}_1.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_1)$ and $\exists\bar{\mathtt{Y}}_2.(\vec{\mathtt{X}} = \vec{\mathtt{T}}_2)$ are equivalent, then there exists a renaming ρ such that $\vec{\mathtt{T}}_1 = \rho(\vec{\mathtt{T}}_2)$. □

Please note that the fact that the canonical solved forms $\exists\bar{\mathtt{Y}}_1.(\vec{\mathtt{X}}_1 = \vec{\mathtt{T}}_1)$ and $\exists\bar{\mathtt{Y}}_2.(\vec{\mathtt{X}}_2 = \vec{\mathtt{T}}_2)$ are equivalent does *not* imply that $\bar{\mathtt{X}}_1$ and $\bar{\mathtt{X}}_2$ coincide. Consider, for example, the canonical solved forms true and $\exists\mathtt{Y}.(\mathtt{X} = \mathtt{Y})$, which by C-NameEq are equivalent. One might wish to further restrict canonical solved forms by requiring $\bar{\mathtt{X}}$ to be the set of *essential* type variables of the constraint $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, that is, the set of the type variables that appear free in *all* equivalent constraints. However, as far our technical development is concerned, it seems more convenient not to do so. Instead, we show that it is possible to explicitly restrict or extend $\bar{\mathtt{X}}$ when needed (Lemma 1.3.43).

The following definition allows entertaining a dual view of canonical solved forms, either as constraints or as idempotent type substitutions. The latter view is commonly found in standard treatments of unification (Lassez, Maher, and Marriott, 1988; Jouannaud and Kirchner, 1991) and in classic presentations of ML-the-type-system.

1.3.41 Definition: If $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ is an idempotent substitution of domain $\bar{\mathtt{X}}$, let $\exists[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ denote the canonical solved form $\exists\bar{\mathtt{Y}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$, where $\bar{\mathtt{Y}} = ftv(\vec{\mathtt{T}})$. An idempotent substitution θ is a *most general unifier* of the constraint C if and only if $\exists\theta$ and C are equivalent. □

By definition, equivalent constraints admit the same most general unifiers. Many properties of canonical solved forms may be reformulated in terms of most general unifiers. By Lemmas 1.3.31, 1.3.35, and 1.3.37, every satisfiable constraint admits a most general unifier. By Lemma 1.3.40, if $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}_1]$ and

$[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}_2]$ are most general unifiers of C, then $\vec{\mathtt{T}}_1$ and $\vec{\mathtt{T}}_2$ coincide up to a renaming. Conversely, if $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]$ is a most general unifier of C and if $\bar{\mathtt{X}} \# \rho$ holds, then $[\vec{\mathtt{X}} \mapsto \rho\vec{\mathtt{T}}]$ is also a most general unifier of C; indeed, these two substitutions correspond to α-equivalent canonical solved forms.

The following result relates the substitution θ to the canonical solved form $\exists\theta$, stating that every ground refinement of the former satisfies the latter.

1.3.42 Lemma: $\theta(\exists\theta) \equiv \mathsf{true}$. □

The following lemma offers two technical results: the domain of a most general unifier of C may be restricted so as to become a subset of $ftv(C)$; it may also be extended to include arbitrary fresh variables. The next lemma is a simple corollary.

1.3.43 Lemma: Let θ be a most general unifier of C. If $\bar{\mathtt{Z}} \# ftv(C)$, then $\theta \setminus \bar{\mathtt{Z}}$ is also a most general unifier of C. If $\bar{\mathtt{Z}} \# \theta$, then there exists a most general unifier of C that extends θ and whose domain is $dom(\theta) \cup \bar{\mathtt{Z}}$. □

1.3.44 Lemma: Let θ_1 and θ_2 be most general unifiers of C. Let $\bar{\mathtt{X}} = dom(\theta_1) \cap dom(\theta_2)$. Then, $\theta_1(\bar{\mathtt{X}})$ and $\theta_2(\bar{\mathtt{X}})$ coincide up to a renaming. □

Our last technical result relates the most general unifiers of C with the most general unifiers of $\exists\mathtt{X}.C$. It states that the former are extensions of the latter. Furthermore, under a few freshness conditions, *every* most general unifier of $\exists\mathtt{X}.C$ may be extended to yield a most general unifier of C.

1.3.45 Lemma: If θ is a most general unifier of C, then $\theta \setminus \mathtt{X}$ is a most general unifier of $\exists\mathtt{X}.C$. Conversely, if θ is a most general unifier of $\exists\mathtt{X}.C$ and $\mathtt{X} \# \theta$ and $ftv(\exists\mathtt{X}.C) \subseteq dom(\theta)$, then there exists a type substitution θ' such that θ' extends θ, θ' is a most general unifier of C, and $dom(\theta') = dom(\theta) \cup \mathtt{X}$. □

1.4 HM(X)

Constraint-based type systems appeared during the 1980s (Mitchell, 1984; Fuh and Mishra, 1988) and were widely studied during the following decade (Curtis, 1990; Aiken and Wimmers, 1993; Jones, 1994a; Smith, 1994; Palsberg, 1995; Trifonov and Smith, 1996; Fähndrich, 1999; Pottier, 2001b). We now present one such system, baptized HM(X) because it is a *parameterized* extension of Hindley and Milner's type discipline; the meaning of the parameter X was explained on page 24. Its original description is due to Odersky, Sulzmann, and Wehr (1999a). Since then, it has been completed in a number of works (Sulzmann, Müller, and Zenger, 1999; Sulzmann, 2000; Pottier, 2001a;

Skalka and Pottier, 2002). Each of these presentations introduces minor variations. Here, we follow (Pottier, 2001a), which is itself inspired by (Sulzmann, Müller, and Zenger, 1999).

Definition

Our presentation of HM(X) relies on the constraint language introduced in section 1.3. Technically, our approach of constraints is more direct than that of (Odersky, Sulzmann, and Wehr, 1999a). We interpret constraints within a model, give conjunction and existential quantification their standard meaning, and derive a number of equivalence laws (Section 1.3). Odersky *et al.*, on the other hand, do not explicitly rely on a logical interpretation; instead, they axiomatize constraint equivalence, that is, they consider a number of equivalence laws as axioms. Thus, they ensure that their high-level proofs, such as type soundness and correctness and completeness of type inference, are independent of the low-level details of the logical interpretation of constraints. Their approach is also more general, since it allows dealing with other logical interpretations—such as "open-world" interpretations, where constraints are interpreted not within a fixed model, but within a *family* of extensions of a "current" model. In this chapter, we have avoided this extra layer of abstraction, for the sake of definiteness; however, the changes required to adopt Odersky *et al.*'s approach would not be extensive, since the forthcoming proofs do indeed rely mostly on constraint equivalence laws, rather than on low-level details of the logical interpretation of constraints.

Another slight departure from Odersky *et al.*'s work lies in the fact that we have enriched the constraint language with type scheme introduction and instantiation forms, which were absent in the original presentation of HM(X). To prevent this addition from affecting HM(X), we require the constraints that appear in HM(X) typing judgements to *have no free program identifiers*. Please note that this does not prevent them from containing let forms; we shall in fact exploit this feature when establishing an equivalence between HM(X) and the type system presented in section 1.5, where the new constraint forms are effectively used.

The type system HM(X) consists of a four-place *judgement* whose parameters are a constraint C, an environment Γ, an expression $\mathtt{t}$, and a type scheme σ. A judgement is written $C, \Gamma \vdash \mathtt{t} : \sigma$ and is read: *under the assumptions C and Γ, the expression $\mathtt{t}$ has type σ*. One may view C as an assumption about the judgement's free type variables and Γ as an assumption about $\mathtt{t}$'s free program identifiers. Please recall that Γ now contains *constrained* type schemes, and that σ is a *constrained* type scheme.

We would like the validity of a typing judgement to depend not on the

$$\frac{\Gamma(\mathtt{x}) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash \mathtt{x} : \sigma} \ (\text{HMX-VAR}) \qquad \frac{C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(C, \Gamma)}{C \wedge \exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}} \ (\text{HMX-GEN})$$

$$\frac{C, (\Gamma; \mathtt{z} : \mathtt{T}) \vdash \mathtt{t} : \mathtt{T}'}{C, \Gamma \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'} \ (\text{HMX-ABS}) \qquad \frac{C, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}}{C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}} \ (\text{HMX-INST})$$

$$\frac{C, \Gamma \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}' \qquad C, \Gamma \vdash \mathtt{t}_2 : \mathtt{T}}{C, \Gamma \vdash \mathtt{t}_1 \ \mathtt{t}_2 : \mathtt{T}'} \ (\text{HMX-APP}) \qquad \frac{C, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad C \Vdash \mathtt{T} \leq \mathtt{T}'}{C, \Gamma \vdash \mathtt{t} : \mathtt{T}'} \ (\text{HMX-SUB})$$

$$\frac{C, \Gamma \vdash \mathtt{t}_1 : \sigma \qquad C, (\Gamma; \mathtt{z} : \sigma) \vdash \mathtt{t}_2 : \mathtt{T}}{C, \Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}} \ (\text{HMX-LET}) \qquad \frac{C, \Gamma \vdash \mathtt{t} : \sigma \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \sigma)}{\exists\bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \sigma} \ (\text{HMX-EXISTS})$$

Figure 1-7: Typing rules for HM(X)

syntax, but only on the *meaning* of its constraint assumption. We enforce this point of view by considering judgements equal modulo equivalence of their constraint assumptions. In other words, the typing judgements $C, \Gamma \vdash \mathtt{t} : \sigma$ and $D, \Gamma \vdash \mathtt{t} : \sigma$ are considered identical when $C \equiv D$ holds. As a result, it does not make sense to analyze the syntax of a judgement's constraint assumption. A judgement is *valid*, or *holds*, if and only if it is derivable via the rules given in Figure 1-7. Please note that a valid judgement may involve an unsatisfiable constraint. A program $\mathtt{t}$ is *well-typed* within the environment Γ if and only if a judgement of the form $C, \Gamma \vdash \mathtt{t} : \sigma$ holds for some *satisfiable* constraint C.

Let us now explain the rules. Like DM-VAR, HMX-VAR looks up the environment to determine the type scheme associated with the program identifier $\mathtt{x}$. The constraint C that appears in the conclusion must be strong enough to guarantee that σ has an instance; this is expressed by the second premise. This technical requirement is used in the proof of Lemma 1.4.1. HMX-ABS, HMX-APP, and HMX-LET are identical to DM-ABS, DM-APP, and DM-LET, respectively, except that the assumption C is made available to every subderivation. We recall that the type $\mathtt{T}$ may be viewed as the type scheme $\forall\varnothing[\mathsf{true}].\mathtt{T}$ (Definitions 1.2.18 and 1.3.2). As a result, types form a subset of type schemes, which implies that $\Gamma; \mathtt{z} : \mathtt{T}$ is a well-formed environment and $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ a well-formed typing judgement. To understand HMX-GEN, it is best to first consider the particular case where C is true. This yields the following, simpler rule:

$$\frac{D, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)}{\exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}} \qquad (\text{HMX-GEN'})$$

The second premise is identical to that of DM-GEN: the type variables that are generalized must not occur free within the environment. The conclusion forms the type scheme $\forall\bar{\mathtt{X}}[D].\mathtt{T}$, where the type variables $\bar{\mathtt{X}}$ have become universally quantified, but are still subject to the constraint D. Please note that the type variables that occur free in D may include not only $\bar{\mathtt{X}}$, but also other type variables, typically free in Γ. The rule's conclusion carries the constraint $\exists\bar{\mathtt{X}}.D$, thus recording the requirement that the newly formed type scheme should have an instance; again, this is used in the proof of Lemma 1.4.1. HMX-GEN may be viewed as a more liberal version of HMX-GEN', whereby part of the current constraint, namely C, need not be copied if it does not concern the type variables that are being generalized, namely $\bar{\mathtt{X}}$. This optimization is important in practice, because C may be very large. An intuitive explanation for its correctness is given by the constraint equivalence law C-LETAND, which expresses the same optimization in terms of let constraints. Because HM(X) does not use let constraints, the optimization is hard-wired into the typing rule. HMX-INST allows taking an instance of a type scheme. The reader may be surprised to find that, contrary to DM-INST, it does not involve a type substitution. Instead, the rule merely drops the universal quantifier, which amounts to applying the identity substitution $\vec{\mathtt{X}} \mapsto \vec{\mathtt{X}}$. One should recall, however, that type schemes are considered equal modulo α-conversion, so it is possible to *rename* the type scheme's universal quantifiers prior to using HMX-INST. The reason why this provides sufficient expressive power appears in the proof of Theorem 1.4.7 below. The constraint D carried by the type scheme is recorded as part of the current constraint in HMX-INST's conclusion. The *subsumption* rule HMX-SUB allows a type $\mathtt{T}$ to be replaced at any time with an arbitrary supertype $\mathtt{T}'$. Because both $\mathtt{T}$ and $\mathtt{T}'$ may have free type variables, whether $\mathtt{T} \leq \mathtt{T}'$ holds depends on the current assumption C, which is why the rule's second premise is an *entailment* assertion. An operational explanation of HMX-SUB is that it requires all uses of subsumption to be explicitly recorded in the current constraint. Please note that HMX-SUB remains a useful and necessary rule even when subtyping is interpreted as equality: then, it allows exploiting the type *equations* found in C. Last, HMX-EXISTS allows the type variables that occur only within the current constraint to become existentially quantified. As a result, these type variables no longer occur free in the rule's conclusion; in other words, they have become *local* to the subderivation rooted at the premise. One may prove that the presence of HMX-EXISTS in the type system does not augment the set of well-typed programs, but does augment the set of valid typing judgements; it is a pleasant technical convenience. Indeed, because judgements are considered equal modulo constraint equivalence, constraints may be transparently *simplified* at any time. (By *simplifying* a constraint, we mean replacing it with an equivalent constraint whose syntactic representation is considered simpler.) Bearing this fact in mind, one finds that an effect of rule HMX-EXISTS is to enable *more* simplifications: because constraint equivalence is a congruence, $C \equiv D$ implies $\exists\bar{\mathtt{X}}.C \equiv \exists\bar{\mathtt{X}}.D$, but the converse does not hold in general. For instance, there is in general no way of simplifying the judgement $\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}, \Gamma \vdash \mathtt{t} : \sigma$, but if it is known that $\mathtt{Y}$ does not appear free in Γ or σ, then HMX-EXISTS allows deriving $\exists\mathtt{Y}.(\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}), \Gamma \vdash \mathtt{t} : \sigma$, which is the same judgement as $\mathtt{X} \leq \mathtt{Z}, \Gamma \vdash \mathtt{t} : \sigma$. Thus, an interesting simplification has been enabled. Please note that $\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z} \equiv \mathtt{X} \leq \mathtt{Z}$ does *not* hold, while, according to C-EXTRANS, $\exists\mathtt{Y}.(\mathtt{X} \leq \mathtt{Y} \leq \mathtt{Z}) \equiv \mathtt{X} \leq \mathtt{Z}$ does.

We now establish a few simple properties of the type system HM(X). Our first lemma is a minor technical property.

1.4.1 LEMMA: $C, \Gamma \vdash \mathtt{t} : \sigma$ implies $C \Vdash \exists\sigma$. □

The next lemma states that *strengthening* a judgement's constraint assumption preserves its validity. In other words, *weakening* a judgement preserves its validity. It is worth noting that in traditional presentations, which rely more heavily on type substitutions, the analogue of this result is a *type substitution* lemma; see for instance (Tofte, 1988, Lemma 2.7), (Leroy, 1992, Proposition 1.2), (Skalka and Pottier, 2002, Lemma 3.4). Here, the lemma further states that weakening a judgement does not alter the shape of its derivation, a useful property when reasoning by induction on type derivations.

1.4.2 LEMMA [WEAKENING]: If $C' \Vdash C$, then every derivation of $C, \Gamma \vdash \mathtt{t} : \sigma$ may be turned into a derivation of $C', \Gamma \vdash \mathtt{t} : \sigma$ with the same shape. □

Proof: The proof is by structural induction on a derivation of $C, \Gamma \vdash \mathtt{t} : \sigma$. In each proof case, we adopt the notations of Figure 1-7.

∘ *Case* HMX-VAR. The rule's conclusion is $C, \Gamma \vdash \mathtt{x} : \sigma$. Its premises are $\Gamma(\mathtt{x}) = \sigma$ **(1)** and $C \Vdash \exists\sigma$ **(2)**. By hypothesis, we have $C' \Vdash C$ **(3)**. By transitivity of entailment, (3) and (2) imply $C' \Vdash \exists\sigma$ **(4)**. By HMX-VAR, (1) and (4) yield $C', \Gamma \vdash \mathtt{x} : \sigma$.

∘ *Cases* HMX-ABS, HMX-APP, HMX-LET. By the induction hypothesis and by HMX-ABS, HMX-APP, or HMX-LET, respectively.

∘ *Case* HMX-GEN. The rule's conclusion is $C \wedge \exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}$. Its premises are $C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(C, \Gamma)$ **(2)**. By hypothesis, we have $C' \Vdash C \wedge \exists\bar{\mathtt{X}}.D$ **(3)**. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(C')$ **(4)**. Applying the induction hypothesis to (1) and to the entailment assertion $C' \wedge C \wedge D \Vdash C \wedge D$, we obtain $C' \wedge C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(5)**. By HMX-GEN, applied to (5), (2) and (4), we get $C' \wedge C \wedge \exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}$ **(6)**. By (3) and C-DUP, the constraints $C' \wedge C \wedge \exists\bar{\mathtt{X}}.D$ and C' are equivalent, so (6) is the goal $C', \Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}[D].\mathtt{T}$.

$\circ$ *Case* HMX-INST. The rule's conclusion is $C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$. Its premise is $C, \Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}[D].\mathtt{T}$ **(1)**. By hypothesis, C' entails $C \wedge D$ **(2)**. Because (2) implies $C' \Vdash C$, the induction hypothesis may be applied to (1), yielding $C', \Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}[D].\mathtt{T}$ **(3)**. By HMX-INST, we obtain $C' \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. Because (2) implies $C' \equiv C' \wedge D$, (4) is the goal $C', \Gamma \vdash \mathtt{t} : \mathtt{T}$.

$\circ$ *Case* HMX-SUB. The rule's conclusion is $C, \Gamma \vdash \mathtt{t} : \mathtt{T}'$. Its premises are $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $C \Vdash \mathtt{T} \leq \mathtt{T}'$ **(2)**. By hypothesis, we have $C' \Vdash C$ **(3)**. Applying the induction hypothesis to (1) and (3) yields $C', \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. By transitivity of entailment, (3) and (2) imply $C' \Vdash \mathtt{T} \leq \mathtt{T}'$ **(5)**. By HMX-SUB, (4) and (5) yield $C', \Gamma \vdash \mathtt{t} : \mathtt{T}'$.

$\circ$ *Case* HMX-EXISTS. The rule's conclusion is $\exists \bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \sigma$. Its premises are $C, \Gamma \vdash \mathtt{t} : \sigma$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \sigma)$ **(2)**. By hypothesis, we have $C' \Vdash \exists \bar{\mathtt{X}}.C$ **(3)**. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(C')$ **(4)**. Applying the induction hypothesis to (1) and to the entailment assertion $C' \wedge C \Vdash C$, we obtain $C' \wedge C, \Gamma \vdash \mathtt{t} : \sigma$ **(5)**. By HMX-EXISTS, (5) and (2) yield $\exists \bar{\mathtt{X}}.(C' \wedge C), \Gamma \vdash \mathtt{t} : \sigma$ **(6)**. By (4) and C-EXAND, the constraint $\exists \bar{\mathtt{X}}.(C' \wedge C)$ is equivalent to $C' \wedge \exists \bar{\mathtt{X}}.C$, which, by (3) and C-DUP, is equivalent to C'. Thus, (6) is the goal $C', \Gamma \vdash \mathtt{t} : \sigma$. $\square$

We do not give a direct type soundness proof for HM(X). Instead, in section 1.5, we prove that it is equivalent to another type system, which later is itself proven sound. A direct type soundness result, based on a denotational semantics, may be found in (Odersky, Sulzmann, and Wehr, 1999a). Another type soundness proof, which follows Wright and Felleisen's syntactic approach (1994b), appears in (Skalka and Pottier, 2002). Last, a hybrid approach, which combines some of the advantages of the previous two, is given in (Pottier, 2001a).

An alternate presentation of HM(X)

The presentation of HM(X) given in Figure 1-7 has only four syntax-directed rules out of eight. It is a good specification of the type system, but it is far from an algorithmic description. As a first step towards such a description, we provide an alternate presentation of HM(X), where generalization is performed only at `let` expressions and instantiation takes place only at references to program identifiers (Figure 1-8). It has the property that all judgements are of the form $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$, rather than $C, \Gamma \vdash \mathtt{t} : \sigma$. The following theorem states that the two presentations are indeed equivalent.

1.4.3 THEOREM: $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ is derivable via the rules of Figure 1-8 if and only if it is a valid HM(X) judgement. $\square$

$$\frac{\Gamma(\mathtt{x}) = \forall\bar{\mathtt{X}}[D].\mathtt{T}}{C \wedge D, \Gamma \vdash \mathtt{x} : \mathtt{T}} \quad (\textsc{hmd-VarInst})$$

$$\frac{C, (\Gamma; \mathtt{z} : \mathtt{T}) \vdash \mathtt{t} : \mathtt{T}'}{C, \Gamma \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'} \quad (\textsc{hmd-Abs})$$

$$\frac{C, \Gamma \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}' \qquad C, \Gamma \vdash \mathtt{t}_2 : \mathtt{T}}{C, \Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'} \quad (\textsc{hmd-App})$$

$$\frac{\begin{array}{c} C \wedge D, \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \bar{\mathtt{X}} \# \mathit{ftv}(C, \Gamma) \\ C \wedge \exists\bar{\mathtt{X}}.D, (\Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[D].\mathtt{T}_1) \vdash \mathtt{t}_2 : \mathtt{T}_2 \end{array}}{C \wedge \exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \quad (\textsc{hmd-LetGen})$$

$$\frac{C, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad C \Vdash \mathtt{T} \leq \mathtt{T}'}{C, \Gamma \vdash \mathtt{t} : \mathtt{T}'} \quad (\textsc{hmd-Sub})$$

$$\frac{C, \Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \mathtt{T})}{\exists\bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \mathtt{T}} \quad (\textsc{hmd-Exists})$$

Figure 1-8: An alternate presentation of HM(X)

This theorem shows that the rule sets of Figures 1-7 and 1-8 derive the same monomorphic judgements, that is, the same judgements of the form $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$. The fact that judgements of the form $C, \Gamma \vdash \mathtt{t} : \sigma$, where σ is a not a monotype, cannot be derived using the new rule set is a technical simplification, without deep significance; the first two exercises below shed some light on this issue.

1.4.4 Exercise [★★]: Show that both rule sets lead to the same set of *well-typed* programs. □

1.4.5 Exercise [★★]: Show that, if hmx-Gen is added to the rule set of Figure 1-8, then both rule sets derive exactly the same judgements. □

1.4.6 Exercise [★★★, $\not\rightarrow$]: Show that it is possible to simplify the presentation of Damas and Milner's type system in an analogous manner. That is, define an alternate set of typing rules for DM, which allows deriving judgements of the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$; then, show that this new rule set is equivalent to the previous one, in the same sense as above. Which auxiliary properties of DM does your proof require? A solution is given in (Clément, Despeyroux, Despeyroux, and Kahn, 1986). □

Relating HM(X) with Damas and Milner's type system

In order to explain our interest in HM(X), we wish to show that it is more general than Damas and Milner's type system. Since HM(X) really is a *family* of type systems, we must make this statement more precise. First, every member of the HM(X) family contains DM. Conversely, DM contains HM($=$), the constraint-based type system obtained by specializing HM(X) to the setting of an equality-only syntactic model.

The first of these assertions is easy to prove, because the mapping from DM judgements to HM(X) judgements is essentially the identity: every valid DM judgement may be viewed as a valid HM(X) judgement under the trivial assumption true. This statement relies on the fact that the DM type scheme $\forall \bar{\mathtt{X}}.\mathtt{T}$ is identified with the constrained type scheme $\forall \bar{\mathtt{X}}[\mathsf{true}].\mathtt{T}$, so DM type schemes (resp. environments) form a subset of HM(X) type schemes (resp. environments). Its proof is routine, except perhaps in the case of DM-INST, where it is shown how the effect of applying a substitution in DM is emulated by strengthening the current constraint in HM(X).

1.4.7 THEOREM: If $\Gamma \vdash \mathtt{t} : \mathtt{S}$ holds in DM, then $\mathsf{true}, \Gamma \vdash \mathtt{t} : \mathtt{S}$ holds in HM(X). □

Proof: The proof is by structural induction on a derivation of $\Gamma \vdash \mathtt{t} : \mathtt{S}$. In each proof case, we adopt the notations of Figure 1-3.

∘ *Case* DM-VAR. The rule's conclusion is $\Gamma \vdash \mathtt{x} : \mathtt{S}$. Its premise is $\Gamma(\mathtt{x}) = \mathtt{S}$ **(1)**. By definition and by C-EX*, the constraint $\exists \mathtt{S}$ is equivalent to true. By applying HMX-VAR to (1) and to the assertion $\mathsf{true} \Vdash \mathsf{true}$, we obtain $\mathsf{true}, \Gamma \vdash \mathtt{x} : \mathtt{S}$.

∘ *Cases* DM-ABS, DM-APP, DM-LET. By the induction hypothesis and by HMX-ABS, HMX-APP or HMX-LET, respectively.

∘ *Case* DM-GEN. The rule's conclusion is $\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}$. Its premises are $\Gamma \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)$ **(2)**. Applying the induction hypothesis to (1) yields $\mathsf{true}, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(3)**. Furthermore, (2) implies $\bar{\mathtt{X}} \# \mathit{ftv}(\mathsf{true}, \Gamma)$ **(4)**. By HMX-GEN, (3) and (4) yield $\mathsf{true}, \Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}[\mathsf{true}].\mathtt{T}$.

∘ *Case* DM-INST. The rule's conclusion is $\Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$. Its premise is $\Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}$ **(1)**. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \bar{\mathtt{T}})$ **(2)**. Applying the induction hypothesis to (1) yields $\mathsf{true}, \Gamma \vdash \mathtt{t} : \forall \bar{\mathtt{X}}[\mathsf{true}].\mathtt{T}$ **(3)**. By HMX-INST, (3) implies $\mathsf{true}, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. By Lemma 1.4.2, we may weaken this judgement so as to obtain $\vec{\mathtt{X}} = \vec{\mathtt{T}}, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(5)**. Using C-EQ, C-EXTRANS, and C-EXAND, it is possible to establish $\vec{\mathtt{X}} = \vec{\mathtt{T}} \Vdash \mathtt{T} = [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(6)**. Applying HMX-SUB to (5) and (6), we find $\vec{\mathtt{X}} = \vec{\mathtt{T}}, \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(7)**. Last, (2) implies $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$ **(8)**. Applying HMX-EXISTS to (7) and (8), we obtain $\exists \bar{\mathtt{X}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}}), \Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$ **(9)**. By (2) and C-NAME, the constraint $\exists \bar{\mathtt{X}}.(\vec{\mathtt{X}} = \vec{\mathtt{T}})$ is equivalent to true, so (9) is the goal. □

We are now interested in proving that HM(=), as defined above, is contained within DM. To this end, we must translate every HM(=) judgement to a DM judgement. It quickly turns out that this is possible if the original judgement's constraint assumption is *satisfiable*.

We begin by explaining how an HM(=) is translated into a DM type scheme. Such a translation is made possible by the fact that the definition of HM(=) assumes an equality-only syntactic model. Indeed, in that setting, every satisfiable constraint admits a most general unifier (Definition 1.3.41), whose properties we make essential use of.

In fact, we must not only translate a type scheme, but also apply a type substitution to it. Instead of separating these steps, we perform both at once, and parameterize the translation by a type substitution θ. (It does not appear that separating them would help.) The definition of $[\![\sigma]\!]_\theta$ is somewhat involved: it is given in the statement of the following lemma, whose proof establishes that the definition is indeed well-formed.

1.4.8 Lemma: Consider a type scheme σ and an idempotent type substitution θ such that $\mathit{ftv}(\sigma) \subseteq \mathit{dom}(\theta)$ **(1)** and $\exists\theta \Vdash \exists\sigma$ **(2)**. Write $\sigma = \forall\bar{\mathtt{X}}[D].\mathtt{T}$, where $\bar{\mathtt{X}} \# \theta$ **(3)**. Then, there exists a type substitution θ' such that θ' extends θ, $\mathit{dom}(\theta')$ is $\mathit{dom}(\theta) \cup \bar{\mathtt{X}}$, and θ' is a most general unifier of $\exists\theta \wedge D$. Let $\bar{\mathtt{Y}} = \mathit{ftv}(\theta'(\bar{\mathtt{X}})) \setminus \mathit{range}(\theta)$. Then, the *translation* of σ under θ, written $[\![\sigma]\!]_\theta$, is the DM type scheme $\forall\bar{\mathtt{Y}}.\theta'(\mathtt{T})$. This is a well-formed definition. Furthermore, $\mathit{ftv}([\![\sigma]\!]_\theta) \subseteq \mathit{range}(\theta)$ holds. □

Proof: By (2), $\exists\theta$ is equivalent to $\exists\theta \wedge \exists\sigma$, which may be written $\exists\theta \wedge \exists\bar{\mathtt{X}}.D$. By (3) and C-ExAnd, this is $\exists\bar{\mathtt{X}}.(\exists\theta \wedge D)$. Thus, because θ is a most general unifier of $\exists\theta$, θ is also a most general unifier of $\exists\bar{\mathtt{X}}.(\exists\theta \wedge D)$ **(4)**. Furthermore, $\mathit{ftv}(\exists\bar{\mathtt{X}}.(\exists\theta \wedge D))$ is $\mathit{ftv}(\exists\theta \wedge \exists\sigma)$, which by definition of $\exists\theta$ and by (1) is a subset of $\mathit{dom}(\theta)$ **(5)**. By (4), (3), (5), and Lemma 1.3.45, there exists a type substitution θ' such that θ' extends θ **(6)** and θ' is a most general unifier of $\exists\theta \wedge D$ **(7)** and $\mathit{dom}(\theta') = \mathit{dom}(\theta) \cup \bar{\mathtt{X}}$ **(8)**.

Let us now define $\bar{\mathtt{Y}} = \mathit{ftv}(\theta'(\bar{\mathtt{X}})) \setminus \mathit{range}(\theta)$ and $[\![\sigma]\!]_\theta = \forall\bar{\mathtt{Y}}.\theta'(\mathtt{T})$. By (1), we have $\mathit{ftv}(\mathtt{T}) \subseteq \bar{\mathtt{X}} \cup \mathit{dom}(\theta)$. Applying θ' and exploiting (6), we find $\mathit{ftv}(\theta'(\mathtt{T})) \subseteq \mathit{ftv}(\theta'(\bar{\mathtt{X}})) \cup \mathit{range}(\theta)$, which by definition of $\bar{\mathtt{Y}}$ may be written $\mathit{ftv}(\theta'(\mathtt{T})) \subseteq \bar{\mathtt{Y}} \cup \mathit{range}(\theta)$. Subtracting $\bar{\mathtt{Y}}$ on each side, we find $\mathit{ftv}([\![\sigma]\!]_\theta) \subseteq \mathit{range}(\theta)$ **(9)**.

To show that the definition of $[\![\sigma]\!]_\theta$ is valid, there remains to show that it does not depend on the choice of $\bar{\mathtt{X}}$ or θ'. To prove the former, it suffices to establish $\bar{\mathtt{X}} \# \mathit{ftv}([\![\sigma]\!]_\theta)$, which indeed follows from (3) and (9). As for the latter, because of the constraints imposed by (6), (7), and (8), and by Lemma 1.3.44, distinct choices of θ' may differ only by a renaming of $\mathit{ftv}(\theta'(\bar{\mathtt{X}})) \setminus \mathit{range}(\theta)$, that is, $\bar{\mathtt{Y}}$. So, we must check $\bar{\mathtt{Y}} \# \mathit{ftv}([\![\sigma]\!]_\theta)$, which holds by definition. □

Please note that if σ is in fact a type $\mathtt{T}$, where $\mathit{ftv}(\mathtt{T}) \subseteq \mathit{dom}(\theta)$, then $\bar{\mathtt{X}}$ is empty, so θ' is θ, $\bar{\mathtt{Y}}$ is empty, and $[\![\mathtt{T}]\!]_\theta = \theta(\mathtt{T})$. In other words, the translation of a type under θ is its image through θ. More generally, the translation of an unconstrained type scheme (that is, a type scheme whose constraint is `true`) is its image through θ, as stated by the following exercise.

1.4.9 Exercise [★★, $\not\rightarrow$]: Prove that $[\![\forall\bar{\mathtt{X}}.\mathtt{T}]\!]_\theta$, when defined, is $\theta(\forall\bar{\mathtt{X}}.\mathtt{T})$. □

The translation becomes more than a mere type substitution when applied to a nontrivial constrained type scheme. Some examples of this situation are given below.

1.4.10 Example: Let $\sigma = \forall\mathtt{XY}[\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Y}].\mathtt{X}$. Let θ be the identity substitution. The type scheme σ is closed and the constraint $\exists\sigma$ is equivalent to `true`, so $[\![\sigma]\!]_\theta$ is defined. We must find a type substitution θ' whose domain is $\mathtt{XY}$ and that is a most general unifier of $\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Y}$. All such substitutions are of the form $[\mathtt{X} \mapsto (\mathtt{Z} \rightarrow \mathtt{Z}), \mathtt{Y} \mapsto \mathtt{Z}]$, where $\mathtt{Z}$ is fresh. We have $\mathit{ftv}(\theta'(\mathtt{XY})) = \mathtt{Z}$, whence $[\![\sigma]\!]_\theta = \forall\mathtt{Z}.\mathtt{Z} \rightarrow \mathtt{Z}$. Note that the choice of $\mathtt{Z}$ does not matter, since it is bound in $[\![\sigma]\!]_\theta$. Roughly speaking, the effect of the translation was to replace the body $\mathtt{X}$ of the constrained type scheme with its most general solution under the constraint $\mathtt{X} = \mathtt{Y} \rightarrow \mathtt{Y}$.

Let $\sigma = \forall\mathtt{X}\mathtt{Y}_1[\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2].\mathtt{X}$. Let $\theta = [\mathtt{Y}_2 \mapsto \mathtt{Z}_2]$. We have $\mathit{ftv}(\sigma) = \mathtt{Y}_2 \subseteq \mathit{dom}(\theta)$. The constraint $\exists\sigma$ is equivalent to `true`, so $[\![\sigma]\!]_\theta$ is defined. We must find a type substitution θ' whose domain is $\mathtt{X}\mathtt{Y}_1\mathtt{Y}_2$ that extends θ and that is a most general unifier of $\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2$. All such substitutions are of the form $[\mathtt{X} \mapsto (\mathtt{Z}_1 \rightarrow \mathtt{Z}_2), \mathtt{Y}_1 \mapsto \mathtt{Z}_1, \mathtt{Y}_2 \mapsto \mathtt{Z}_2]$, where $\mathtt{Z}_1$ is fresh. We have $\mathit{ftv}(\theta'(\mathtt{X}\mathtt{Y}_1)) \setminus \mathit{range}(\theta) = \mathtt{Z}_1\mathtt{Z}_2 \setminus \mathtt{Z}_2 = \mathtt{Z}_1$, whence $[\![\sigma]\!]_\theta = \forall\mathtt{Z}_1.\mathtt{Z}_1 \rightarrow \mathtt{Z}_2$. The type variable $\mathtt{Z}_2$ is *not* universally quantified—even though it appears in the image of $\mathtt{X}$, which *was* universally quantified in σ—because $\mathtt{Z}_2$ is the image of $\mathtt{Y}_2$, which was free in σ. □

Before attacking the main theorem, let us establish a couple of technical properties of the translation. First, $[\![\sigma]\!]_\theta$ is insensitive to the behavior of θ outside $\mathit{ftv}(\sigma)$, a natural property, since our informal intent is for θ to be applied to σ.

1.4.11 Lemma: If θ_1 and θ_2 coincide on $\mathit{ftv}(\sigma)$, then $[\![\sigma]\!]_{\theta_1}$ and $[\![\sigma]\!]_{\theta_2}$ are either both undefined, or both defined and identical. □

Second, if $C \Vdash \sigma \preceq \mathtt{T}'$ holds, then the translations of σ and $\mathtt{T}'$ under a most general unifier of C are in Damas and Milner's instance relation. One might say, roughly speaking, that the instance relation is preserved by the translation.

1.4.12 Lemma: Let $\mathit{ftv}(\sigma, \mathtt{T}') \subseteq \mathit{dom}(\theta)$ **(1)** and $\exists\theta \Vdash \exists\sigma$ **(2)**. Let $\exists\theta \Vdash \sigma \preceq \mathtt{T}'$ **(3)**. Then, $\theta(\mathtt{T}')$ is an instance of the DM type scheme $[\![\sigma]\!]_\theta$. □

Proof: Write $\sigma = \forall\bar{\mathtt{X}}[D].\mathtt{T}$, where $\bar{\mathtt{X}} \# \theta$ **(4)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}')$ **(5)**. By (1), (2), and (4), one may define θ', $\bar{\mathtt{Y}}$, and $[\![\sigma]\!]_\theta$ exactly as in the statement of Lemma 1.4.8. By (5) and Definition 1.3.3, (3) is synonymous with $\exists\theta \Vdash \exists\bar{\mathtt{X}}.(D\wedge$

$\mathtt{T} = \mathtt{T}'$). Reasoning in the same manner as in the first paragraph of the proof of Lemma 1.4.8, we find that there exists a type substitution θ'' such that θ'' extends θ, $dom(\theta'')$ is $dom(\theta) \cup \bar{\mathtt{X}}$, and θ'' is a most general unifier of $\exists\theta \wedge D \wedge \mathtt{T} = \mathtt{T}'$.

We have $dom(\theta') = dom(\theta'')$ **(6)**. Furthermore, θ' is a most general unifier of $\exists\theta \wedge D$, while θ'' is a most general unifier of $\exists\theta \wedge D \wedge \mathtt{T} = \mathtt{T}'$, which implies $\exists\theta'' \Vdash \exists\theta'$ **(7)**. By Lemma 1.3.39, θ'' refines θ'. That is, there exists a type substitution φ such that θ'' is the restriction of $\varphi \circ \theta'$ to $dom(\theta) \cup \bar{\mathtt{X}}$ **(8)**. We may require $dom(\varphi) \subseteq range(\theta) \cup ftv(\theta'(\bar{\mathtt{X}}))$ **(9)** without compromising (8).

Consider $\mathtt{X} \in dom(\theta)$. Because θ'' extends θ, we have $\theta''(\mathtt{X}) = \theta(\mathtt{X})$ **(10)**. Furthermore, by (8), we have $\theta''(\mathtt{X}) = (\varphi \circ \theta')(\mathtt{X}) = (\varphi \circ \theta)(\mathtt{X})$ **(11)**. Using (10) and (11), we find $\theta(\mathtt{X}) = \varphi(\theta(\mathtt{X}))$. Because this holds for every $\mathtt{X} \in dom(\theta)$, φ must be the identity over $range(\theta)$; that is, $dom(\varphi) \# range(\theta)$ **(12)** holds. Combining (9) and (12), we find $dom(\varphi) \subseteq ftv(\theta'(\bar{\mathtt{X}})) \setminus range(\theta)$, that is, $dom(\varphi) \subseteq \bar{\mathtt{Y}}$ **(13)**.

By construction of θ'', we have $\exists\theta'' \Vdash \mathtt{T} = \mathtt{T}'$. By Lemma 1.3.29, this implies $\theta''(\exists\theta'') \Vdash \theta''(\mathtt{T}) = \theta''(\mathtt{T}')$, which by Lemma 1.3.42 may be read $\mathsf{true} \Vdash \theta''(\mathtt{T}) = \theta''(\mathtt{T}')$. By Lemma 1.3.32, $\theta''(\mathtt{T})$ and $\theta''(\mathtt{T}')$ coincide. Because by (1) $ftv(\mathtt{T})$ is a subset of $dom(\theta) \cup \bar{\mathtt{X}}$ and by (8), the former may be written $\varphi(\theta'(\mathtt{T}))$. By (1) and because θ'' extends θ, the latter is $\theta(\mathtt{T}')$. Thus, we have $\varphi(\theta'(\mathtt{T})) = \theta(\mathtt{T}')$. Together with (13), this establishes that $\theta(\mathtt{T}')$ is an instance of $\forall\bar{\mathtt{Y}}.\theta'(\mathtt{T})$, that is, $[\![\sigma]\!]_\theta$. □

We extend the translation to environments as follows. $[\![\varnothing]\!]_\theta$ is $\varnothing$. If $\exists\theta \Vdash \exists\sigma$ holds, then $[\![\Gamma; \mathtt{x} : \sigma]\!]_\theta$ is $[\![\Gamma]\!]_\theta; \mathtt{x} : [\![\sigma]\!]_\theta$, otherwise it is $[\![\Gamma]\!]_\theta$. Notice that $[\![\Gamma]\!]_\theta$ contains fewer bindings than Γ, which ensures that bindings $x : \sigma$ for which $\exists\theta \Vdash \exists\sigma$ does not hold will not be used in the translation. Please note that $[\![\Gamma]\!]_\theta$ is defined when $ftv(\Gamma) \subseteq dom(\theta)$ holds.

We are now ready to prove the main theorem. Please note that, by requiring θ to be a most general unifier of C, we also require C to be satisfiable. Judgements that carry an unsatisfiable constraint cannot be translated.

1.4.13 Theorem: Let $C, \Gamma \vdash \mathtt{t} : \sigma$ hold in HM(=). Let θ be a most general unifier of C such that $ftv(\Gamma, \sigma) \subseteq dom(\theta)$. Then, $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : [\![\sigma]\!]_\theta$ holds in DM. □

Proof: Let us first remark that, by Lemma 1.4.1, we have $C \Vdash \exists\sigma$. This may be written $\exists\theta \Vdash \exists\sigma$, which guarantees that $[\![\sigma]\!]_\theta$ is defined. The proof is by structural induction on an HM(=) typing derivation. We assume that the derivation is expressed in terms of the rules of Figure 1-8, but split hmd-LetGen into hmx-Let and hmx-Gen for the sake of readability.

∘ *Case* hmd-VarInst. The rule's conclusion is $C \wedge D, \Gamma \vdash \mathtt{x} : \mathtt{T}$. By hypothesis, θ is a most general unifier of $C \wedge D$ **(1)**, and $ftv(\mathtt{T}) \subseteq dom(\theta)$ **(2)**

holds. The rule's premise is $\Gamma(\mathtt{x}) = \sigma$ **(3)**, where σ stands for $\forall\bar{\mathtt{X}}[D].\mathtt{T}$. By (1), we have $\exists\theta \equiv C \wedge D \Vdash D \Vdash \exists\bar{\mathtt{X}}.D \equiv \exists\sigma$ **(4)**. Furthermore, we have $\mathit{ftv}(\sigma) \subseteq \mathit{ftv}(\Gamma) \subseteq \mathit{dom}(\theta)$ **(5)**. These facts show that $[\![\sigma]\!]_\theta$ is defined. Together with (3), this implies $[\![\Gamma]\!]_\theta(\mathtt{x}) = [\![\sigma]\!]_\theta$. By DM-VAR, $[\![\Gamma]\!]_\theta \vdash \mathtt{x} : [\![\sigma]\!]_\theta$ **(6)** follows. Now, by Lemma 1.3.19, we have $D \Vdash \sigma \preceq \mathtt{T}$, which, combined with $\exists\theta \Vdash D$, yields $\exists\theta \Vdash \sigma \preceq \mathtt{T}$ **(7)**. By (7), (4), (5), (2), and Lemma 1.4.12, we find that $\theta(\mathtt{T})$ is an instance of $[\![\sigma]\!]_\theta$. Thus, applying DM-INST to (6) yields $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : \theta(\mathtt{T})$.

∘ *Case* HMD-ABS. The rule's conclusion is $C, \Gamma \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'$. Its premise is $C, (\Gamma; \mathtt{z} : \mathtt{T}) \vdash \mathtt{t} : \mathtt{T}'$. Applying the induction hypothesis to it yields $[\![\Gamma]\!]_\theta; \mathtt{z} : \theta(\mathtt{T}) \vdash \mathtt{t} : \theta(\mathtt{T}')$. By DM-ABS, this implies $[\![\Gamma]\!]_\theta \vdash \lambda\mathtt{z}.\mathtt{t} : \theta(\mathtt{T}) \rightarrow \theta(\mathtt{T}')$, that is, $[\![\Gamma]\!]_\theta \vdash \lambda\mathtt{z}.\mathtt{t} : \theta(\mathtt{T} \rightarrow \mathtt{T}')$.

∘ *Case* HMD-APP. By an extension of $\mathit{dom}(\theta)$ to include $\mathit{ftv}(\mathtt{T})$, by the induction hypothesis, and by DM-APP.

∘ *Case* HMX-LET. By an extension of $\mathit{dom}(\theta)$ to include $\mathit{ftv}(\sigma)$, by the induction hypothesis, and by DM-LET.

∘ *Case* HMX-GEN. The rule's conclusion is $C \wedge \exists\sigma, \Gamma \vdash \mathtt{t} : \sigma$, where σ stands for $\forall\bar{\mathtt{X}}[D].\mathtt{T}$. By hypothesis, θ is a most general unifier of $C \wedge \exists\sigma$ **(1)**, and $\mathit{ftv}(\Gamma, \sigma) \subseteq \mathit{dom}(\theta)$ **(2)** holds. The rule's premises are $C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(3)** and $\bar{\mathtt{X}} \mathrel{\#} \mathit{ftv}(C, \Gamma)$ **(4)**. We may further assume, *w.l.o.g.*, $\bar{\mathtt{X}} \mathrel{\#} \theta$ **(5)**. Given (1), (2), and (5), we may define θ' and $\bar{\mathtt{Y}}$ exactly as in Lemma 1.4.8. Then, θ' is a most general unifier of $\exists\theta \wedge D$, that is, $C \wedge D$. Furthermore, $\mathit{dom}(\theta')$ is $\mathit{dom}(\theta) \cup \bar{\mathtt{X}}$, which by (2) is a superset of $\mathit{ftv}(\Gamma, \mathtt{T})$. Thus, the induction hypothesis applies to θ' and to (3), yielding $[\![\Gamma]\!]_{\theta'} \vdash \mathtt{t} : \theta'(\mathtt{T})$. Because θ' extends θ, by (2) and by Lemma 1.4.11, this may be read $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : \theta'(\mathtt{T})$ **(6)**. According to Lemma 1.4.8, we have $\mathit{ftv}([\![\Gamma]\!]_\theta) \subseteq \mathit{range}(\theta)$, which by construction of $\bar{\mathtt{Y}}$ implies $\bar{\mathtt{Y}} \mathrel{\#} \mathit{ftv}([\![\Gamma]\!]_\theta)$ **(7)**. By DM-GEN, (6) and (7) yield $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : \forall\bar{\mathtt{Y}}.\theta'(\mathtt{T})$, that is, $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : [\![\sigma]\!]_\theta$.

∘ *Case* HMD-SUB. The rule's conclusion is $C, \Gamma \vdash \mathtt{t} : \mathtt{T}'$. By hypothesis, θ is a most general unifier of C **(1)**, and $\mathit{ftv}(\Gamma, \mathtt{T}') \subseteq \mathit{dom}(\theta)$ **(2)** holds. The goal is $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : \theta(\mathtt{T}')$ **(3)**. The rule's premises are $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)** and $C \Vdash \mathtt{T} = \mathtt{T}'$ **(5)**. We may assume, *w.l.o.g.*, $\mathit{ftv}(\mathtt{T}) \mathrel{\#} \mathit{range}(\theta)$ **(6)**. Then, by (6) and Lemma 1.3.43, we may extend the domain of θ, so as to achieve $\mathit{ftv}(\mathtt{T}) \subseteq \mathit{dom}(\theta)$ **(7)**, without compromising (1) or (2) or affecting the goal (3). By (1), (2), and (7), the induction hypothesis applies to (4), yielding $[\![\Gamma]\!]_\theta \vdash \mathtt{t} : \theta(\mathtt{T})$ **(8)**. Now, thanks to (1), (5) may be read $\exists\theta \Vdash \mathtt{T} = \mathtt{T}'$, which by Lemmas 1.3.29 and 1.3.42 implies $\mathtt{true} \Vdash \theta(\mathtt{T}) = \theta(\mathtt{T}')$. Then, Lemma 1.3.32 shows that $\theta(\mathtt{T})$ and $\theta(\mathtt{T}')$ coincide. As a result, (8) is the goal (3).

∘ *Case* HMD-EXISTS. The rule's conclusion is $\exists\bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \mathtt{T}$. By hypothesis, θ is a most general unifier of $\exists\bar{\mathtt{X}}.C$ **(1)**, and $\mathit{ftv}(\Gamma, \mathtt{T}) \subseteq \mathit{dom}(\theta)$ **(2)** holds. The rule's premises are $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(3)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \mathtt{T})$. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \theta$ **(4)**. As in the previous case, we may extend the domain of θ to guarantee $\mathit{ftv}(\exists\bar{\mathtt{X}}.C) \subseteq \mathit{dom}(\theta)$ **(5)**. By (1), (4), (5), and Lemma 1.3.45, there exists a type substitution θ' such that θ' extends θ **(6)** and θ' is a most general unifier of C. Applying the induction hypothesis to θ' and to (3) yields $[\![\Gamma]\!]_{\theta'} \vdash \mathtt{t} : \theta'(\mathtt{T})$. By (2), (6), and Lemma 1.4.11, this may be read $[\![\Gamma]\!]_{\theta} \vdash \mathtt{t} : \theta(\mathtt{T})$. □

Together, Theorems 1.4.7 and 1.4.13 yield a precise correspondence between DM and HM(=): there exists a compositional translation from each to the other. In other words, they may be viewed as two equivalent formulations of a single type system. One might also say that HM(=) is a constraint-based formulation of DM. Furthermore, Theorem 1.4.7 states that every member of the HM(X) family is an extension of DM. This explains our double interest in HM(X), as an alternate formulation of DM, which we believe is more pleasant, for reasons already discussed, and as a more expressive framework.

1.5 A purely constraint-based type system: PCB(X)

In the previous section, we have presented HM(X), an elegant constraint-based extension of Damas and Milner's type system. However, HM(X), as described there, suffers from a drawback. A typing judgement involves both a constraint, which represents an assumption about its free type variables, and an environment, which represents an assumption about its free program identifiers. At a let node, HMD-LETGEN turns a part of the current constraint, namely D, into a type scheme, namely $\forall\bar{\mathtt{X}}[D].\mathtt{T}$, and stores it into the environment. Then, at every occurrence of the `let`-bound variable, HMD-VARINST retrieves this type scheme from the environment and adds a copy of D back to the current constraint. In practice, it is important to *simplify* the type scheme $\forall\bar{\mathtt{X}}[D].\mathtt{T}$ *before* it is stored in the environment, because it would be inefficient to copy an unsimplified constraint. In other words, it appears that, in order to preserve efficiency, constraint generation and constraint simplification cannot be separated.

Of course, in practice, it is not difficult to intermix these phases, so the problem is not technical, but pedagogical. Indeed, we argued earlier that it is natural and desirable to separate them. *Type scheme introduction and elimination constraints*, which we introduced in Section 1.3 but did not use in the specification of HM(X), are intended as a means of solving this problem. In the present section, we exploit them to give a novel formulation of HM(X), which no longer requires copying constraints back and forth between the environment and the constraint assumption. In fact, the environment is

$$\frac{C \Vdash \mathtt{x} \preceq \mathtt{T}}{C \vdash \mathtt{x} : \mathtt{T}} \quad (\textsc{Var})$$

$$\frac{C \vdash \mathtt{t} : \mathtt{T}'}{\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda \mathtt{z}.\mathtt{t} : \mathtt{T} \to \mathtt{T}'} \quad (\textsc{Abs})$$

$$\frac{C_1 \vdash \mathtt{t}_1 : \mathtt{T} \to \mathtt{T}' \qquad C_2 \vdash \mathtt{t}_2 : \mathtt{T}}{C_1 \wedge C_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'} \quad (\textsc{App})$$

$$\frac{C_1 \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad C_2 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \quad (\textsc{Let})$$

$$\frac{C \vdash \mathtt{t} : \mathtt{T}}{C \wedge \mathtt{T} \leq \mathtt{T}' \vdash \mathtt{t} : \mathtt{T}'} \quad (\textsc{Sub})$$

$$\frac{C \vdash \mathtt{t} : \mathtt{T} \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})}{\exists \bar{\mathtt{X}}.C \vdash \mathtt{t} : \mathtt{T}} \quad (\textsc{Exists})$$

Figure 1-9: Typing rules for PCB(X)

suppressed altogether: taking advantage of the new constraint forms, we encode information about program identifiers within the constraint assumption.

Presentation

We now employ the full constraint language (Section 1.3). Typing judgements take the form $C \vdash \mathtt{t} : \mathtt{T}$, where C may have free type variables *and* free program identifiers. The rules that allow deriving such judgements appear in Figure 1-9. As before, we identify judgements up to constraint equivalence.

Let us review the rules. Var states that $\mathtt{x}$ has type $\mathtt{T}$ under any constraint that entails $\mathtt{x} \preceq \mathtt{T}$. Note that we no longer consult the type scheme associated with $\mathtt{x}$ in the environment—indeed, there is no environment. Instead, we let the constraint assumption record the fact that the type scheme should admit $\mathtt{T}$ as one of its instances. Thus, in a judgement $C \vdash \mathtt{t} : \mathtt{T}$, any program identifier that occurs free within $\mathtt{t}$ typically also occurs free within C. Abs requires the body $\mathtt{t}$ of a λ-abstraction to have type $\mathtt{T}'$ under assumption C. Although no explicit assumption about $\mathtt{z}$ appears in the premise, C typically contains a number of instantiation constraints bearing on $\mathtt{z}$, of the form $\mathtt{z} \preceq \mathtt{T}_i$. In the rule's conclusion, C is wrapped within the prefix $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ []$, where $\mathtt{T}$ is the type assigned to $\mathtt{z}$. This effectively requires every $\mathtt{T}_i$ to denote a supertype of $\mathtt{T}$, as desired. Please note that $\mathtt{z}$ does not occur free in the constraint $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C$, which is natural, since it does not occur free in $\lambda \mathtt{z}.\mathtt{t}$. App exhibits a minor stylistic difference with respect to hmx-App: its constraint assumption is split between its premises. It is not difficult to prove that, when weakening holds (see Lemma 1.5.2 below), this choice does not affect the set of valid judgements. This new presentation encourages reading the rules in Figure 1-9 as the specification of an algorithm, which, given $\mathtt{t}$ and $\mathtt{T}$, produces C such that $C \vdash \mathtt{t} : \mathtt{T}$ holds. In the case of App, the algorithm invokes itself recursively for each of the two subexpressions, yielding the constraints C_1 and C_2, then constructs their conjunction. Let is analogous to Abs: by wrapping C_2 within a let prefix, it gives meaning to the instantiation constraints bearing on $\mathtt{z}$ within C_2. The difference is that $\mathtt{z}$ may now be assigned a type scheme, as opposed to a monotype. An appropriate type scheme is built in the most straightforward manner from the constraint C_1 and the type $\mathtt{T}_1$ that describe $\mathtt{t}_1$. All of the type variables that appear free in the left-hand premise are generalized, hence the notation $\forall\mathcal{V}[C_1].\mathtt{T}_1$, which is a convenient shorthand for $\forall \mathit{ftv}(C_1, \mathtt{T}_1)[C_1].\mathtt{T}_1$. The side-condition that "type variables that occur free in the environment must not be generalized", which was present in DM and HM(X), naturally disappears, since judgements no longer involve an environment. Sub again exhibits a minor stylistic difference with respect to hmx-Sub: the comments made about App above apply here as well. Exists is essentially identical to hmx-Exists.

In the standard specification of HM(X), hmd-Abs and hmd-LetGen accumulate information in the environment. Through the environment, this information is made available to hmd-VarInst, which retrieves and copies it. Here, instead, no information is explicitly transmitted. Where a program identifier is bound, a type scheme introduction constraint is built; where a program identifier is used, a type scheme instantiation constraint is produced. The two are related only by our definition of the meaning of constraints.

The reader may be puzzled by the fact that Let allows *all* type variables that occur free in its left-hand premise to be generalized. The following exercise sheds some light on this issue.

1.5.1 Exercise [★, Recommended]: Build a type derivation for the expression $\lambda\mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ within PCB(X). Draw a comparison with the solution of Exercise 1.2.21. □

The following lemma is an analogue of Lemma 1.4.2.

1.5.2 Lemma [Weakening]: If $C' \Vdash C$, then every derivation of $C \vdash \mathtt{t} : \mathtt{T}$ may be turned into a derivation of $C' \vdash \mathtt{t} : \mathtt{T}$ with the same shape. □

Proof: The proof is by structural induction on a derivation of $C \vdash \mathtt{t} : \mathtt{T}$. In each proof case, we adopt the notations of Figure 1-9.

∘ *Case* Var. By transitivity of entailment.

∘ *Case* Abs. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \to \mathtt{T}'$ **(1)**. By hypothesis, we have $C' \Vdash \mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C$ **(2)**. We may assume, *w.l.o.g.*, $\mathtt{z} \notin \mathit{fpi}(C')$ **(3)**. The rule's premise is $C \vdash \mathtt{t} : \mathtt{T}'$ **(4)**. Applying the induction hypothesis to (4) yields $C \wedge C' \vdash \mathtt{t} : \mathtt{T}'$, which by Abs implies $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ (C \wedge$

C') $\vdash \lambda \mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}'$ **(5)**. By (3) and C-InAnd*, $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ (C \wedge C')$ is equivalent to $(\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C) \wedge C'$, which by (2) and C-Dup is equivalent to C'. Thus, (5) is the goal $C' \vdash \lambda \mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}'$.

∘ *Case* App. By applying the induction hypothesis to each premise, using the fact that $C' \Vdash C_1 \wedge C_2$ implies $C' \Vdash C_1$ and $C' \Vdash C_2$.

∘ *Case* Let. Analogous to the case of Abs. The induction hypothesis is applied to the second premise only.

∘ *Case* Sub. Analogous to the case of App.

∘ *Case* Exists. See the corresponding case in the proof of Lemma 1.4.2. □

Relating PCB(X) with HM(X)

Let us now provide evidence for our claim that PCB(X) is an alternate presentation of HM(X). The next two theorems define an effective translation from HM(X) to PCB(X) and back.

The first theorem states that if, within HM(X), $\mathtt{t}$ has type $\mathtt{T}$ under assumptions C and Γ, then, within PCB(X), $\mathtt{t}$ also has type $\mathtt{T}$, under some assumption C'. The relationship $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ states that C entails the residual constraint obtained by confronting Γ, which provides information about the free program identifiers in $\mathtt{t}$, with C', which contains instantiation constraints bearing on these program identifiers. The statement requires C and Γ to have no free program identifiers, which is natural, since they are part of an HM(X) judgement. The hypothesis $C \Vdash \exists\Gamma$ excludes the somewhat pathological situation where Γ contains constraints not apparent in C. This hypothesis vanishes when Γ is the initial environment; see Definition 1.7.3.

1.5.3 Theorem: Let $C \Vdash \exists\Gamma$. Assume $fpi(C, \Gamma) = \varnothing$. If $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ holds in HM(X), then there exists a constraint C' such that $C' \vdash \mathtt{t} : \mathtt{T}$ holds in PCB(X) and C entails $\mathsf{let}\ \Gamma\ \mathsf{in}\ C'$. □

Proof: The proof is by structural induction on a derivation of $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$. In each proof case, we adopt the notations of Figure 1-8.

∘ *Case* hmd-VarInst. The rule's conclusion is $C \wedge D, \Gamma \vdash \mathtt{x} : \mathtt{T}$. By hypothesis, we have $C \wedge D \Vdash \exists\Gamma$ **(1)** and $fpi(C, D, \Gamma) = \varnothing$ **(2)**. The rule's premise is $\Gamma(\mathtt{x}) = \forall\bar{\mathtt{X}}[D].\mathtt{T}$ **(3)**. By Var, we have $\mathtt{x} \preceq \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$, so there remains to establish $C \wedge D \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}$ **(4)**. By (3), (2), and C-InId, the constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}$ is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ \forall\bar{\mathtt{X}}[D].\mathtt{T} \preceq \mathtt{T}$, which, by (2) and C-In*, is itself equivalent to $\exists\Gamma \wedge \forall\bar{\mathtt{X}}[D].\mathtt{T} \preceq \mathtt{T}$ **(5)**. By (1) and Lemma 1.3.19, $C \wedge D$ entails (5). We have established (4).

∘ *Case* hmd-Abs. The rule's conclusion is $C, \Gamma \vdash \lambda \mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}'$. Its premise is $C, (\Gamma; \mathtt{z} : \mathtt{T}) \vdash \mathtt{t} : \mathtt{T}'$ **(1)**. The constraints $\exists\Gamma$ and $\exists(\Gamma; \mathtt{z} : \mathtt{T})$ are equivalent, so the induction hypothesis applies to (1) and yields a constraint C' such that $C' \vdash \mathtt{t} : \mathtt{T}'$ **(2)** and $C \Vdash \mathsf{let}\ \Gamma; \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C'$ **(3)**. Applying Abs to (2) yields $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C' \vdash \lambda\mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}'$. There remains to check that C entails $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C'$—but that is precisely (3).

∘ *Case* hmd-App. The rule's conclusion is $C, \Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'$. Its premises are $C, \Gamma \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}'$ **(1)** and $C, \Gamma \vdash \mathtt{t}_2 : \mathtt{T}$ **(2)**. Applying the induction hypothesis to (1) and (2), we obtain constraints C'_1 and C'_2 such that $C'_1 \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}'$ **(3)** and $C'_2 \vdash \mathtt{t}_2 : \mathtt{T}$ **(4)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'_1$ **(5)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'_2$ **(6)**. By App, (3) and (4) imply $C'_1 \wedge C'_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'$. Furthermore, by C-InAnd, (5) and (6) yield $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'_1 \wedge C'_2$.

∘ *Case* hmd-LetGen. The rule's conclusion is $C \wedge \exists\bar{\mathtt{X}}.D, \Gamma \vdash \mathtt{let\ z = t_1\ in\ t_2} : \mathtt{T}_2$. By hypothesis, we have $C \wedge \exists\bar{\mathtt{X}}.D \Vdash \exists\Gamma$ **(1)** and $\mathit{fpi}(C, D, \Gamma) = \varnothing$ **(2)**. The rule's premises are $C \wedge D, \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(3)** and $\bar{\mathtt{X}} \# \mathit{ftv}(C, \Gamma)$ **(4)** and $C \wedge \exists\bar{\mathtt{X}}.D, \Gamma' \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(5)**, where Γ' is $\Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[D].\mathtt{T}_1$. Applying the induction hypothesis to (3) yields a constraint C'_1 such that $C'_1 \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(6)** and $C \wedge D \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'_1$ **(7)**. By (1), (2), and C-In*, we have $C \wedge \exists\bar{\mathtt{X}}.D \Vdash \exists\Gamma'$. Thus, the induction hypothesis applies to (5) and yields a constraint C'_2 such that $C'_2 \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(8)** and $C \wedge \exists\bar{\mathtt{X}}.D \Vdash \mathsf{let}\ \Gamma'\ \mathsf{in}\ C'_2$ **(9)**. By Let, (6) and (8) imply $\mathsf{let}\ \mathtt{z} : \forall\mathcal{V}[C'_1].\mathtt{T}_1\ \mathsf{in}\ C'_2 \vdash \mathtt{let\ z = t_1\ in\ t_2} : \mathtt{T}_2$ **(10)**. By Lemmas 1.3.25 and 1.5.2, (10) yields $\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}[C'_1].\mathtt{T}_1\ \mathsf{in}\ C'_2 \vdash \mathtt{let\ z = t_1\ in\ t_2} : \mathtt{T}_2$ **(11)**, where the universal quantification is over $\bar{\mathtt{X}}$ only. There remains to establish that $C \wedge \exists\bar{\mathtt{X}}.D$ entails $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[C'_1].\mathtt{T}_1\ \mathsf{in}\ C'_2$ **(12)**. By (4), (2), and C-LetDup, the constraint (12) is equivalent to $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C'_1].\mathtt{T}_1\ \mathsf{in}\ C'_2$. By (7), this constraint is entailed by $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[C \wedge D].\mathtt{T}_1\ \mathsf{in}\ C'_2$, which by (4) and C-LetAnd, is equivalent to $C \wedge \mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[D].\mathtt{T}_1\ \mathsf{in}\ C'_2$, that is, $C \wedge \mathsf{let}\ \Gamma'\ \mathsf{in}\ C'_2$. By (9), this constraint is entailed by $C \wedge \exists\bar{\mathtt{X}}.D$.

∘ *Case* hmd-Sub. The rule's conclusion is $C, \Gamma \vdash \mathtt{t} : \mathtt{T}'$. Its premises are $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $C \Vdash \mathtt{T} \leq \mathtt{T}'$ **(2)**. Applying the induction hypothesis to (1) yields a constraint C' such that $C' \vdash \mathtt{t} : \mathtt{T}$ **(3)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ **(4)**. By Sub, (3) implies $C' \wedge \mathtt{T} \leq \mathtt{T}' \vdash \mathtt{t} : \mathtt{T}'$. There remains to establish $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ (C' \wedge \mathtt{T} \leq \mathtt{T}')$, which follows from (4) and (2) by C-InAnd*.

∘ *Case* hmd-Exists. The rule's conclusion is $\exists\bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \mathtt{T}$. Its premises are $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \mathtt{T})$ **(2)**. By hypothesis, we have $\exists\bar{\mathtt{X}}.C \Vdash \exists\Gamma$, which by Lemma 1.3.16 implies $C \Vdash \exists\Gamma$. Thus, the induction hypothesis applies to (1) and yields a constraint C' such that $C' \vdash \mathtt{t} : \mathtt{T}$ **(3)** and $C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ C'$ **(4)**. By Exists, (3) and (2) imply $\exists\bar{\mathtt{X}}.C' \vdash \mathtt{t} : \mathtt{T}$. There remains to establish $\exists\bar{\mathtt{X}}.C \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.C'$. By congruence of entailment, (4) implies $\exists\bar{\mathtt{X}}.C \Vdash \exists\bar{\mathtt{X}}.\mathsf{let}\ \Gamma\ \mathsf{in}\ C'$. The result follows by (2) and C-InEx. □

The second theorem states that if, within PCB(X), $\mathtt{t}$ has type $\mathtt{T}$ under assumption C, then, within HM(X), $\mathtt{t}$ also has type $\mathtt{T}$, under assumptions

$\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ and Γ. The idea is simple: the constraint C represents a combined assumption about the initial judgement's free type variables and free program identifiers. In HM(X), these two kinds of assumptions must be maintained separately. So, we split them into a pair of an environment Γ, which may be chosen arbitrarily, provided it satisfies $fpi(C) \subseteq dpi(\Gamma)$—that is, provided it defines all program variables of interest, and the residual constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$, which has no free program identifiers, thus represents an assumption about the new judgement's type variables only. Distinct choices of Γ give rise to distinct HM(X) judgements, which may be incomparable; this is related to the fact that ML-the-type-system does not have principal typings (Jim, 1995). Again, the hypothesis $fpi(\Gamma) = fpi(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) = \varnothing$ is natural, since we wish Γ and $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ to appear in an HM(X) judgement.

1.5.4 THEOREM: Assume $fpi(\Gamma) = fpi(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) = \varnothing$ and $C \not\equiv \mathsf{false}$. If $C \vdash \mathtt{t} : \mathtt{T}$ holds in PCB(X), then $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ holds in HM(X). □

Proof: The proof is by structural induction on a derivation of $C \vdash \mathtt{t} : \mathtt{T}$. In each proof case, we adopt the notations of Figure 1-9.

By Lemma 1.3.30, the hypothesis $C \not\equiv \mathsf{false}$ is preserved whenever the induction hypothesis is invoked. It is explicitly used only in case VAR, where it guarantees that the identifier at hand is bound in Γ.

∘ *Case* VAR. The rule's conclusion is $C \vdash \mathtt{x} : \mathtt{T}$. Its premise is $C \Vdash \mathtt{x} \preceq \mathtt{T}$ **(1)**. By Lemma 1.3.24, (1) and the hypothesis $C \not\equiv \mathsf{false}$ imply $\mathtt{x} \in fpi(C)$. Because $\mathsf{let}\ \Gamma\ \mathsf{in}\ C$ has no free program identifiers, this implies $\mathtt{x} \in dpi(\Gamma)$, that is, the environment Γ must define $\mathtt{x}$. Let $\Gamma(\mathtt{x}) = \forall\bar{\mathtt{X}}[D].\mathtt{T}'$ **(2)**, where $\bar{\mathtt{X}} \# ftv(\Gamma, \mathtt{T})$ **(3)**. By (2), HMD-VARINST, and HMD-SUB, we have $D \wedge \mathtt{T}' \leq \mathtt{T}, \Gamma \vdash \mathtt{x} : \mathtt{T}$. By (3) and HMD-EXISTS, this implies $\exists\bar{\mathtt{X}}.(D \wedge \mathtt{T}' \leq \mathtt{T}), \Gamma \vdash \mathtt{x} : \mathtt{T}$ **(4)**. Now, by (3), the constraint $\exists\bar{\mathtt{X}}.(D \wedge \mathtt{T}' \leq \mathtt{T})$ may be written $\forall\bar{\mathtt{X}}[D].\mathtt{T}' \preceq \mathtt{T}$ **(5)**. The hypothesis $fpi(\Gamma) = \varnothing$ implies $fpi(D) = \varnothing$ **(6)**. By (6), C-INID and C-IN*, (5) is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}$. Thus, (4) may be written $\mathsf{let}\ \Gamma\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T}, \Gamma \vdash \mathtt{x} : \mathtt{T}$. By (1), by congruence of entailment, and by Lemma 1.4.2, this implies $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash \mathtt{x} : \mathtt{T}$.

∘ *Case* ABS. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}'$ **(1)**. Let Γ' stand for $\Gamma; \mathtt{z} : \mathtt{T}$. Applying the induction hypothesis to (1) yields $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C, \Gamma' \vdash \mathtt{t} : \mathtt{T}'$. By HMD-ABS, this implies $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C, \Gamma \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'$.

∘ *Case* APP. The rule's conclusion is $C_1 \wedge C_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'$. Its premises are $C_1 \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}'$ and $C_2 \vdash \mathtt{t}_2 : \mathtt{T}$. Applying the induction hypothesis yields respectively $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1, \Gamma \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}'$ and $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_2, \Gamma \vdash \mathtt{t}_2 : \mathtt{T}$, which by Lemma 1.4.2 and HMD-APP imply $\mathsf{let}\ \Gamma\ \mathsf{in}\ (C_1 \wedge C_2), \Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'$.

∘ *Case* LET. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \forall\mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ \mathtt{z} =$

$\mathtt{t}_1$ $\mathtt{in}$ $\mathtt{t}_2 : \mathtt{T}_2$. Its premises are $C_1 \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(1)** and $C_2 \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(2)**. Let $\bar{\mathtt{X}}$ stand for $\mathit{ftv}(C_1, \mathtt{T}_1)$. We may require, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, C_2)$ **(3)**. By hypothesis, we have $\mathit{fpi}(\Gamma) = \varnothing$ **(4)**. We also have $\mathit{fpi}(\mathsf{let}\ \Gamma; \mathtt{z} : \forall\mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2) = \varnothing$, which implies $\mathit{fpi}(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1) = \varnothing$. Thus, the induction hypothesis applies to (1) and yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1, \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(5)**. Now, let σ stand for $\forall\bar{\mathtt{X}}[\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1].\mathtt{T}_1$ and Γ' stand for $\Gamma; \mathtt{z} : \sigma$. We have $\mathit{fpi}(\Gamma') = \mathit{fpi}(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) = \varnothing$. Thus, the induction hypothesis applies to (2) and yields $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2, \Gamma' \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(6)**. Let us now weaken (5) and (6) so as to make them suitable premises for HMD-LetGen. Applying Lemma 1.4.2 to (5) yields $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge (\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(7)**. Applying Lemma 1.4.2 to (6) yields $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{\mathtt{X}}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma' \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(8)**. Last, (3) implies $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma, \mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2)$ **(9)**. Applying HMD-LetGen to (7), (9) and (8), we obtain $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{\mathtt{X}}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1), \Gamma \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2$ **(10)**. Now, by (4), (3), and C-LetDup, $\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2$ is equivalent to $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2$. Using this fact, as well as (3), C-InEx, and C-InAnd, we find that the constraint $(\mathsf{let}\ \Gamma'\ \mathsf{in}\ C_2) \wedge \exists\bar{\mathtt{X}}.(\mathsf{let}\ \Gamma\ \mathsf{in}\ C_1)$ is equivalent to $\mathsf{let}\ \Gamma\ \mathsf{in}\ (\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \wedge \exists\bar{\mathtt{X}}.C_1)$, which by definition of the let form, is itself equivalent to $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2$. Last, by definition of $\bar{\mathtt{X}}$, this constraint is $\mathsf{let}\ \Gamma; \mathtt{z} : \forall\mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2$. Thus, (10) is the goal.

∘ *Case* Sub. The rule's conclusion is $C \wedge \mathtt{T} \leq \mathtt{T}' \vdash \mathtt{t} : \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}$ **(1)**. Applying the induction hypothesis to (1) yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(2)**. By Lemma 1.4.2 and HMD-Sub, (2) implies $(\mathsf{let}\ \Gamma\ \mathsf{in}\ C) \wedge \mathtt{T} \leq \mathtt{T}', \Gamma \vdash \mathtt{t} : \mathtt{T}'$, which by C-InAnd* may be written $\mathsf{let}\ \Gamma\ \mathsf{in}\ (C \wedge \mathtt{T} \leq \mathtt{T}'), \Gamma \vdash \mathtt{t} : \mathtt{T}'$.

∘ *Case* Exists. The rule's conclusion is $\exists\bar{\mathtt{X}}.C \vdash \mathtt{t} : \mathtt{T}$. Its premises are $C \vdash \mathtt{t} : \mathtt{T}$ **(1)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$ **(2)**. We may further require, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)$ **(3)**. Applying the induction hypothesis to (1) yields $\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. Applying HMD-Exists to (2), (3), and (4), we find $\exists\bar{\mathtt{X}}.\mathsf{let}\ \Gamma\ \mathsf{in}\ C, \Gamma \vdash \mathtt{t} : \mathtt{T}$, which, by (3) and C-InEx, may be written $\mathsf{let}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.C, \Gamma \vdash \mathtt{t} : \mathtt{T}$. □

As a corollary, we find that, for closed programs, the type systems HM(X) and PCB(X) coincide. In particular, a program is well-typed with respect to one if and only if it is well-typed with respect to the other. This supports the view that PCB(X) is an alternate formulation of HM(X).

1.5.5 Theorem: Assume $\mathit{fpi}(C) = \varnothing$ and $C \not\equiv \mathsf{false}$. Then, $C, \varnothing \vdash \mathtt{t} : \mathtt{T}$ holds in HM(X) if and only if $C \vdash \mathtt{t} : \mathtt{T}$ holds in PCB(X). □

1.6 Constraint generation

We now explain how to reduce type inference problems for PCB(X) to constraint solving problems. A type inference problem consists of an expression

$$\begin{array}{rcl}
[\![\mathtt{x} : \mathtt{T}]\!] & = & \mathtt{x} \preceq \mathtt{T} \\
[\![\lambda \mathsf{z}.\mathtt{t} : \mathtt{T}]\!] & = & \exists \mathtt{X}_1 \mathtt{X}_2.(\mathsf{let}\ \mathsf{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}_2]\!] \wedge \mathtt{X}_1 \rightarrow \mathtt{X}_2 \leq \mathtt{T}) \\
[\![\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}]\!] & = & \exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \rightarrow \mathtt{T}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!]) \\
[\![\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] & = & \mathsf{let}\ \mathsf{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!]
\end{array}$$

Figure 1-10: Constraint generation

t and a type T of kind $\star$. The problem is to determine whether t is well-typed with type T, that is, whether there exists a satisfiable constraint C such that $C \vdash \mathtt{t} : \mathtt{T}$ holds. This formulation of the problem may seem to require an appropriate type T to be known in advance; this is not really the case, since T may be a type variable. A constraint solving problem consists of a constraint C. The problem is to determine whether C is satisfiable. To reduce a type inference problem $(\mathtt{t}, \mathtt{T})$ to a constraint solving problem, we must produce a constraint C that is both *sufficient* and *necessary* for $C \vdash \mathtt{t} : \mathtt{T}$ to hold. Below, we explain how to compute such a constraint, which we write $[\![\mathtt{t} : \mathtt{T}]\!]$. We check that it is indeed *sufficient* by proving $[\![\mathtt{t} : \mathtt{T}]\!] \vdash \mathtt{t} : \mathtt{T}$. That is, the constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ is specific enough to guarantee that t has type T. We say that constraint generation is *sound*. We check that it is indeed *necessary* by proving that, for every constraint C, $C \vdash \mathtt{t} : \mathtt{T}$ implies $C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$. That is, every constraint that guarantees that t has type T is at least as specific as $[\![\mathtt{t} : \mathtt{T}]\!]$. We say that constraint generation is *complete*. Together, these properties mean that $[\![\mathtt{t} : \mathtt{T}]\!]$ is the *least specific* constraint that guarantees that t has type T.

We now see how to reduce a type inference problem to a constraint solving problem. Indeed, if there exists a satisfiable constraint C such that $C \vdash \mathtt{t} : \mathtt{T}$ holds, then, by the completeness property, $C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$ holds, so $[\![\mathtt{t} : \mathtt{T}]\!]$ is satisfiable. Conversely, the soundness property states that $[\![\mathtt{t} : \mathtt{T}]\!] \vdash \mathtt{t} : \mathtt{T}$ holds, so, if $[\![\mathtt{t} : \mathtt{T}]\!]$ is satisfiable, then there exists a satisfiable constraint C such that $C \vdash \mathtt{t} : \mathtt{T}$ holds. In other words, t is well-typed with type T if and only if $[\![\mathtt{t} : \mathtt{T}]\!]$ is satisfiable.

The existence of such a constraint is the analogue of the existence of *principal type schemes* in classic presentations of ML-the-type-system (Damas and Milner, 1982). Indeed, a principal type scheme is least specific in the sense that all valid types are substitution instances of it. Here, the constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ is least specific in the sense that all valid constraints entail it. Earlier, we have established a connection between constraint entailment and refinement of type substitutions, in the specific case of equality constraints interpreted over a free algebra of finite types; see Lemma 1.3.39.

The constraint $[\![\mathtt{t} : \mathtt{T}]\!]$ is defined in Figure 1-10 by induction on the structure of the expression $\mathtt{t}$. We refer to these defining equations as the *constraint generation rules.* The definition is quite terse. It is perhaps even simpler than the declarative specification of $\mathrm{PCB}(X)$ given in Figure 1-9; yet, we prove below that the two are equivalent.

Before explaining the definition, we state the requirements that bear on the type variables $\mathtt{X}_1$, $\mathtt{X}_2$, and $\mathtt{X}$, which appear bound in the right-hand sides of the second, third, and fourth equations. These type variables must have kind $\star$. They must be chosen distinct (that is, $\mathtt{X}_1 \neq \mathtt{X}_2$ in the second equation) and *fresh* in the following sense: *type variables that appear bound in an equation's right-hand side must not appear free in the equation's left-hand side.* Provided this restriction is obeyed, different choices of $\mathtt{X}_1$, $\mathtt{X}_2$, and $\mathtt{X}$ lead to α-equivalent constraints—that is, to the same constraint, since we identify objects up to α-conversion—which guarantees that the above equations make sense. We remark that, since expressions do not have free type variables, the freshness requirement may be simplified to: type variables that appear bound in an equation's right-hand side must not appear free in $\mathtt{T}$. However, this simplification is rendered invalid by the introduction of type annotations within expressions (page 102). Please note that we are able to state a *formal* freshness requirement. This is made possible by the fact that $[\![\mathtt{t} : \mathtt{T}]\!]$ has no free type variables other than those of $\mathtt{T}$, which in turn depends on our explicit use of existential quantification.

Let us now review the four equations. The first one simply mirrors VAR. The second one requires $\mathtt{t}$ to have type $\mathtt{X}_2$ under the hypothesis that $\mathtt{z}$ has type $\mathtt{X}_1$, and forms the arrow type $\mathtt{X}_1 \rightarrow \mathtt{X}_2$; this corresponds to ABS. Here, $\mathtt{X}_1$ and $\mathtt{X}_2$ must be fresh type variables, because we cannot in general guess the expected types of $\mathtt{z}$ and $\mathtt{t}$. The expected type $\mathtt{T}$ is required to be a supertype of $\mathtt{X}_1 \rightarrow \mathtt{X}_2$; this corresponds to SUB. We must bind the fresh type variables $\mathtt{X}_1$ and $\mathtt{X}_2$, so as to guarantee that the generated constraint is unique up to α-conversion. Furthermore, we must bind them *existentially*, because we intend the constraint solver to choose some appropriate value for them. This is justified by EXISTS. The third equation uses the fresh type variable $\mathtt{X}_2$ to stand for the unknown type of $\mathtt{t}_2$. The subexpression $\mathtt{t}_1$ is expected to have type $\mathtt{X}_2 \rightarrow \mathtt{T}$. This corresponds to APP. The fourth equation, which corresponds to LET, is most interesting. It summons a fresh type variable $\mathtt{X}$ and produces $[\![\mathtt{t}_1 : \mathtt{X}]\!]$. This constraint, whose sole free type variable is $\mathtt{X}$, is the *least specific* constraint that must be imposed on $\mathtt{X}$ so as to make it a valid type for $\mathtt{t}_1$. As a result, the type scheme $\forall\mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}$, abbreviated σ in the following, is a *principal* type scheme for $\mathtt{t}_1$. There remains to place $[\![\mathtt{t}_2 : \mathtt{T}]\!]$ inside the context $\mathsf{let}\ \mathtt{z} : \sigma\ \mathsf{in}\ []$. Indeed, when placed inside this context, an instantiation constraint of the form $\mathtt{z} \preceq \mathtt{T}'$ acquires the meaning

$\sigma \preceq \mathtt{T}'$, which by definition of σ and by Lemma 1.6.4 (see below) is equivalent to $[\![\mathtt{t}_1 : \mathtt{T}']\!]$. Thus, the constraint produced by the fourth equation simulates a textual expansion of the `let` construct, whereby every occurrence of `z` would be replaced with $\mathtt{t}_1$. Thanks to type scheme introduction and instantiation constraints, however, this effect is achieved without duplication of source code or constraints. In other words, constraint generation has linear time and space complexity; duplication may take place during constraint solving only.

1.6.1 Exercise [★, ↛]: Define the *size* of an expression, of a type, and of a constraint, viewed as abstract syntax trees. Check that the size of $[\![\mathtt{t} : \mathtt{T}]\!]$ is linear in the sum of the sizes of `t` and `T`. □

We now establish several properties of constraint generation. We begin with soundness, whose proof is straightforward.

1.6.2 Theorem [Soundness]: $[\![\mathtt{t} : \mathtt{T}]\!] \vdash \mathtt{t} : \mathtt{T}$. □

Proof: By induction on the structure of `t`.

∘ *Case* `x`. The goal $\mathtt{x} \preceq \mathtt{T} \vdash \mathtt{x} : \mathtt{T}$ follows from Var.

∘ *Case* $\lambda\mathtt{z.t}$. By the induction hypothesis, we have $[\![\mathtt{t} : \mathtt{X}_2]\!] \vdash \mathtt{t} : \mathtt{X}_2$. By Abs, this implies $\mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}_2]\!] \vdash \lambda\mathtt{z.t} : \mathtt{X}_1 \to \mathtt{X}_2$. By Sub, this implies $\mathsf{let}\ \mathtt{z} : \mathtt{X}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}_2]\!] \wedge \mathtt{X}_1 \to \mathtt{X}_2 \leq \mathtt{T} \vdash \lambda\mathtt{z.t} : \mathtt{T}$. Lastly, because $\mathtt{X}_1\mathtt{X}_2 \# \mathit{ftv}(\mathtt{T})$ holds, Exists applies and yields $[\![\lambda\mathtt{z.t} : \mathtt{T}]\!] \vdash \lambda\mathtt{z.t} : \mathtt{T}$.

∘ *Case* $\mathtt{t}_1\ \mathtt{t}_2$. By the induction hypothesis, we have $[\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \vdash \mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}$ and $[\![\mathtt{t}_2 : \mathtt{X}_2]\!] \vdash \mathtt{t}_2 : \mathtt{X}_2$. By App, this implies $[\![\mathtt{t}_1 : \mathtt{X}_2 \to \mathtt{T}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!] \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}$. Because $\mathtt{X}_2 \notin \mathit{ftv}(\mathtt{T})$ holds, Exists applies and yields $[\![\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}]\!] \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}$.

∘ *Case* $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$. By the induction hypothesis, we have $[\![\mathtt{t}_1 : \mathtt{X}]\!] \vdash \mathtt{t}_1 : \mathtt{X}$ and $[\![\mathtt{t}_2 : \mathtt{T}]\!] \vdash \mathtt{t}_2 : \mathtt{T}$. By Let, these imply $\mathsf{let}\ \mathtt{z} : \forall\mathcal{V}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}$. Because $\mathit{ftv}([\![\mathtt{t}_1 : \mathtt{X}]\!])$ is `X`, the universal quantification on $\mathcal{V}$ really bears on `X` alone. We have proved $[\![\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}$. □

The following lemmas are used in the proof of the completeness property and in a number of other occasions. The first two state that $[\![\mathtt{t} : \mathtt{T}]\!]$ is *covariant* with respect to `T`. Roughly speaking, this means that enough subtyping constraints are generated to achieve completeness with respect to Sub.

1.6.3 Lemma: $[\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}'$ entails $[\![\mathtt{t} : \mathtt{T}']\!]$. □

1.6.4 Lemma: $\mathtt{X} \notin \mathit{ftv}(\mathtt{T})$ implies $\exists\mathtt{X}.([\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{T}) \equiv [\![\mathtt{t} : \mathtt{T}]\!]$. □

The next lemma gives a simplified version of the second constraint generation rule, in the specific case where the expected type is an arrow type. Then, fresh type variables need not be generated; one may directly use the arrow's domain and codomain instead.

1.6.5 LEMMA: $[\![\lambda \mathtt{z.t} : \mathtt{T}_1 \rightarrow \mathtt{T}_2]\!]$ is equivalent to $\mathsf{let}\ \mathtt{z} : \mathtt{T}_1\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}_2]\!]$. □

We conclude with the completeness property.

1.6.6 THEOREM [COMPLETENESS]: if $C \vdash \mathtt{t} : \mathtt{T}$, then $C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$. □

Proof: By induction on the derivation of $C \vdash \mathtt{t} : \mathtt{T}$.

∘ *Case* VAR. The rule's conclusion is $C \vdash \mathtt{x} : \mathtt{T}$. Its premise is $C \Vdash \mathtt{x} \preceq \mathtt{T}$, which is also the goal.

∘ *Case* ABS. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda \mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}'$. By the induction hypothesis, we have $C \Vdash [\![\mathtt{t} : \mathtt{T}']\!]$. By congruence of entailment, this implies $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \Vdash \mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}']\!]$, which, by Lemma 1.6.5, may be written $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \Vdash [\![\lambda \mathtt{z.t} : \mathtt{T} \rightarrow \mathtt{T}']\!]$.

∘ *Case* APP. The rule's conclusion is $C_1 \wedge C_2 \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}'$. Its premises are $C_1 \vdash \mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}'$ and $C_2 \vdash \mathtt{t}_2 : \mathtt{T}$. By the induction hypothesis, we have $C_1 \Vdash [\![\mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}']\!]$ and $C_2 \Vdash [\![\mathtt{t}_2 : \mathtt{T}]\!]$. Thus, $C_1 \wedge C_2$ entails $[\![\mathtt{t}_1 : \mathtt{T} \rightarrow \mathtt{T}']\!] \wedge [\![\mathtt{t}_2 : \mathtt{T}]\!]$, which, by C-NAMEEQ, may be written $\exists \mathtt{X}_2.(\mathtt{X}_2 = \mathtt{T} \wedge [\![\mathtt{t}_1 : \mathtt{X}_2 \rightarrow \mathtt{T}']\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!])$, where $\mathtt{X}_2 \notin \mathit{ftv}(\mathtt{T}, \mathtt{T}')$. Forgetting about the equation $\mathtt{X}_2 = \mathtt{T}$, we find that $C_1 \wedge C_2$ entails $\exists \mathtt{X}_2.([\![\mathtt{t}_1 : \mathtt{X}_2 \rightarrow \mathtt{T}']\!] \wedge [\![\mathtt{t}_2 : \mathtt{X}_2]\!])$, which is precisely $[\![\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}']\!]$.

∘ *Case* LET. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \vdash \mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2$. Its premises are $C_1 \vdash \mathtt{t}_1 : \mathtt{T}_1$ and $C_2 \vdash \mathtt{t}_2 : \mathtt{T}_2$. By the induction hypothesis, we have $C_1 \Vdash [\![\mathtt{t}_1 : \mathtt{T}_1]\!]$ and $C_2 \Vdash [\![\mathtt{t}_2 : \mathtt{T}_2]\!]$, which implies $\mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \Vdash \mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[[\![\mathtt{t}_1 : \mathtt{T}_1]\!]].\mathtt{T}_1\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}_2]\!]$ **(1)**.

Now, let us establish $\mathsf{true} \Vdash \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X} \preceq \forall \mathcal{V}[[\![\mathtt{t}_1 : \mathtt{T}_1]\!]].\mathtt{T}_1$ **(2)**. By definition, this requires proving $\exists \bar{\mathtt{X}}_1.([\![\mathtt{t}_1 : \mathtt{T}_1]\!] \wedge \mathtt{T}_1 \leq \mathtt{Z}) \Vdash \exists \mathtt{X}.([\![\mathtt{t}_1 : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{Z})$ **(3)**, where $\bar{\mathtt{X}}_1 = \mathit{ftv}(\mathtt{T}_1)$ and $\mathtt{Z} \notin \mathtt{X}\bar{\mathtt{X}}_1$ **(4)**. By Lemma 1.6.3, (4), and C-EX*, the left-hand side of (3) entails $[\![\mathtt{t}_1 : \mathtt{Z}]\!]$. By (4) and Lemma 1.6.4, the right-hand side of (3) is $[\![\mathtt{t}_1 : \mathtt{Z}]\!]$. Thus, (3) holds, and so does (2).

By (2) and Lemma 1.3.22, we have $\mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[[\![\mathtt{t}_1 : \mathtt{T}_1]\!]].\mathtt{T}_1\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}_2]\!] \Vdash \mathsf{let}\ \mathtt{z} : \forall \mathtt{X}[[\![\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}_2]\!]$ **(5)**. By transitivity of entailment, (1) and (5) yield $\mathsf{let}\ \mathtt{z} : \forall \mathcal{V}[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \Vdash [\![\mathtt{let\ z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2]\!]$.

∘ *Case* SUB. The rule's conclusion is $C \wedge \mathtt{T} \leq \mathtt{T}' \vdash \mathtt{t} : \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}$. By the induction hypothesis, we have $C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$, which implies $C \wedge \mathtt{T} \leq \mathtt{T}' \Vdash [\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}'$. By lemma 1.6.3 and by transitivity of entailment, we obtain $C \wedge \mathtt{T} \leq \mathtt{T}' \Vdash [\![\mathtt{t} : \mathtt{T}']\!]$.

∘ *Case* EXISTS. The rule's conclusion is $\exists \bar{\mathtt{X}}.C \vdash \mathtt{t} : \mathtt{T}$. Its premises are $C \vdash \mathtt{t} : \mathtt{T}$ and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$ **(1)**. By the induction hypothesis, we have $C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$.

By congruence of entailment, this implies $\exists\bar{\mathtt{X}}.C \Vdash \exists\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!]$ **(2)**. Furthermore, (1) implies $\bar{\mathtt{X}} \# \mathit{ftv}([\![\mathtt{t} : \mathtt{T}]\!])$ **(3)**. By (3) and C-Ex*, (2) may be written $\exists\bar{\mathtt{X}}.C \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$. □

1.7 Type soundness

We are now ready to establish type soundness for our type system. The statement that we wish to prove is sometimes known as *Milner's slogan*: *well-typed programs do not go wrong* (Milner, 1978). Below, we define well-typedness in terms of our constraint generation rules, for the sake of convenience, and establish type soundness with respect to that particular definition. Theorems 1.4.7, 1.5.4, and 1.6.6 imply that type soundness also holds when well-typedness is defined with respect to the typing judgements of DM, HM(X), or PCB(X). We establish type soundness by following Wright and Felleisen's so-called *syntactic approach* (1994b). The approach consists in isolating two independent properties. *Subject reduction*, whose exact statement will be given below, implies that well-typedness is preserved by reduction. *Progress* states that no stuck configuration is well-typed. It is immediate to check that, if both properties hold, then no well-typed program can reduce to a stuck configuration. Subject reduction itself depends on a key lemma, usually known as a (term) *substitution lemma*. We immediately give two versions of this lemma: the former is stated in terms of PCB(X) judgements, while the latter is stated in terms of the constraint generation rules.

1.7.1 LEMMA [SUBSTITUTION]: $C \vdash \mathtt{t} : \mathtt{T}$ and $C_0 \vdash \mathtt{t}_0 : \mathtt{T}_0$ imply $\mathsf{let}\ \mathsf{z}_0 : \forall\bar{\mathtt{X}}_0[C_0].\mathtt{T}_0\ \mathsf{in}\ C \vdash [\mathsf{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$. □

Proof: The proof is by structural induction on the derivation of $C \vdash \mathtt{t} : \mathtt{T}$. In each proof case, we adopt the notations of Figure 1-9. We write σ_0 for $\forall\bar{\mathtt{X}}_0[C_0].\mathtt{T}_0$. We refer to the hypothesis $C_0 \vdash \mathtt{t}_0 : \mathtt{T}_0$ as **(1)**. We assume, *w.l.o.g.*, $\bar{\mathtt{X}}_0 \# \mathit{ftv}(C, \mathtt{T})$ **(2)** and $\mathsf{z}_0 \notin \mathit{fpi}(\sigma_0)$ **(3)**.

∘ *Case* VAR. The rule's conclusion is $C \vdash \mathtt{x} : \mathtt{T}$ **(4)**. Its premise is $C \Vdash \mathtt{x} \preceq \mathtt{T}$ **(5)**. Two subcases arise.

Subcase x is z_0. Applying SUB to (1) yields $C_0 \wedge \mathtt{T}_0 \leq \mathtt{T} \vdash \mathtt{t}_0 : \mathtt{T}$. By (2) and EXISTS, this implies $\exists\bar{\mathtt{X}}_0.(C_0 \wedge \mathtt{T}_0 \leq \mathtt{T}) \vdash \mathtt{t}_0 : \mathtt{T}$ **(6)**. Furthermore, by (2) again, the constraint $\exists\bar{\mathtt{X}}_0.(C_0 \wedge \mathtt{T}_0 \leq \mathtt{T})$ is $\sigma_0 \preceq \mathtt{T}$, which is equivalent to $\mathsf{let}\ \mathsf{z}_0 : \sigma_0\ \mathsf{in}\ \mathsf{z}_0 \preceq \mathtt{T}$. As a result, (6) may be written $\mathsf{let}\ \mathsf{z}_0 : \sigma_0\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T} \vdash [\mathsf{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$ **(7)**.

Subcase x isn't z_0. Then, $[\mathsf{z}_0 \mapsto \mathtt{t}_0]\mathtt{x}$ is x. Thus, VAR yields $\exists\sigma_0 \wedge \mathtt{x} \preceq \mathtt{T} \vdash [\mathsf{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$. By C-IN*, this may be read $\mathsf{let}\ \mathsf{z}_0 : \sigma_0\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T} \vdash [\mathsf{z}_0 \mapsto \mathtt{t}_0]\mathtt{x} : \mathtt{T}$, that is, again (7).

In either subcase, by (5), by congruence of entailment, and by Lemma 1.5.2, (7) implies $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$.

∘ *Case* Abs. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C \vdash \lambda\mathtt{z}.\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'$. Its premise is $C \vdash \mathtt{t} : \mathtt{T}'$ **(8)**. We may assume, *w.l.o.g.*, that $\mathtt{z}$ is distinct from $\mathtt{z}_0$ and does not occur free within $\mathtt{t}_0$ or σ_0 **(9)**. Applying the induction hypothesis to (8) yields $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}'$, which, by Abs, implies $\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ (\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C) \vdash \lambda\mathtt{z}.[\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T} \rightarrow \mathtt{T}'$. By (9) and C-LetLet, this may be written $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ (\mathsf{let}\ \mathtt{z} : \mathtt{T}\ \mathsf{in}\ C) \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0](\lambda\mathtt{z}.\mathtt{t}) : \mathtt{T} \rightarrow \mathtt{T}'$.

∘ *Case* App. By the induction hypothesis, by App, and by C-InAnd.

∘ *Case* Let. The rule's conclusion is $\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}_1[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \vdash \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2$, where $\bar{\mathtt{X}}_1 = \mathit{ftv}(C_1, \mathtt{T}_1)$. Its premises are $C_1 \vdash \mathtt{t}_1 : \mathtt{T}_1$ **(10)** and $C_2 \vdash \mathtt{t}_2 : \mathtt{T}_2$ **(11)**. We may assume, *w.l.o.g.*, that $\mathtt{z}$ is distinct from $\mathtt{z}_0$ and does not occur free within $\mathtt{t}_0$ or σ_0 **(12)**. We may also assume, *w.l.o.g.*, $\bar{\mathtt{X}}_1 \# \mathit{ftv}(\sigma_0)$ **(13)**. Applying the induction hypothesis to (10) and (11) respectively yields $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_1 \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t}_1 : \mathtt{T}_1$ **(14)** and $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_2 \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t}_2 : \mathtt{T}_2$ **(15)**. Applying Let to (14) and (15) produces $\mathsf{let}\ \mathtt{z} : \forall\mathcal{V}[\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_1].\mathtt{T}_1\ \mathsf{in}\ \mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_2 \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0](\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2) : \mathtt{T}_2$ **(16)**. Now, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \mathtt{z}_0 : \sigma_0; \mathtt{z} : \forall\bar{\mathtt{X}}_1[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 & \\
\equiv & \mathsf{let}\ \mathtt{z}_0 : \sigma_0; \mathtt{z} : \forall\bar{\mathtt{X}}_1[\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 & \mathbf{(17)} \\
\equiv & \mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}_1[\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_1].\mathtt{T}_1; \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_2 & \mathbf{(18)} \\
\Vdash & \mathsf{let}\ \mathtt{z} : \forall\mathcal{V}\ [\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_1].\mathtt{T}_1; \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C_2 & \mathbf{(19)}
\end{array}$$

where (17) follows from (13), (3), and C-LetDup; (18) follows from (12) and C-LetLet; and (19) is by Lemma 1.3.25. Thus, applying Lemma 1.5.2 to (16) yields $\mathsf{let}\ \mathtt{z}_0 : \sigma_0; \mathtt{z} : \forall\bar{\mathtt{X}}_1[C_1].\mathtt{T}_1\ \mathsf{in}\ C_2 \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0](\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2) : \mathtt{T}_2$.

∘ *Case* Sub. By the induction hypothesis, by Sub, and by C-InAnd*.

∘ *Case* Exists. The rule's conclusion is $\exists\bar{\mathtt{X}}.C \vdash \mathtt{t} : \mathtt{T}$. Its premises are $C \vdash \mathtt{t} : \mathtt{T}$ **(20)** and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$ **(21)**. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\sigma_0)$ **(22)**. Applying the induction hypothesis to (20) yields $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$, which, by (21) and Exists, implies $\exists\bar{\mathtt{X}}.\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$ **(23)**. By (22) and C-InEx, (23) is $\mathsf{let}\ \mathtt{z}_0 : \sigma_0\ \mathsf{in}\ \exists\bar{\mathtt{X}}.C \vdash [\mathtt{z}_0 \mapsto \mathtt{t}_0]\mathtt{t} : \mathtt{T}$. □

1.7.2 Lemma: $\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}[\llbracket \mathtt{t}_2 : \mathtt{T}_2 \rrbracket].\mathtt{T}_2\ \mathsf{in}\ \llbracket \mathtt{t}_1 : \mathtt{T}_1 \rrbracket$ entails $\llbracket [\mathtt{z} \mapsto \mathtt{t}_2]\mathtt{t}_1 : \mathtt{T}_1 \rrbracket$. □

Before going on, let us give a few definitions and formulate several requirements. First, we must define an *initial environment* Γ_0, which assigns a type scheme to every constant. A couple of requirements must be made to ensure that Γ_0 is consistent with the semantics of constants, as specified by $\xrightarrow{\delta}$. Second, we must extend constraint generation and well-typedness to *configurations*, as opposed to programs, since reduction operates on configurations.

Last, we must formulate a *restriction* to tame the interaction between side effects and `let`-polymorphism, which is unsound if unrestricted.

1.7.3 Definition: Let Γ_0 be an environment whose domain is the set of constants $\mathcal{Q}$. We require $ftv(\Gamma_0) = \varnothing$, $fpi(\Gamma_0) = \varnothing$, and $\exists\Gamma_0 \equiv \mathsf{true}$. We refer to Γ_0 as the *initial* typing environment. □

1.7.4 Definition: Let ref be an isolated, invariant type constructor of signature $\star \Rightarrow \star$. A *store type* M is a finite mapping from memory locations to types. We write $\mathsf{ref}\, M$ for the environment that maps m to $\mathsf{ref}\, M(m)$ when m is in the domain of M. Assuming $dom(\mu)$ and $dom(M)$ coincide, the constraint $[\![\mu : M]\!]$ is defined as the conjunction of the constraints $[\![\mu(m) : M(m)]\!]$, where m ranges over $dom(\mu)$. Under the same assumption, the constraint $[\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$ is defined as $[\![\mathtt{t} : \mathtt{T}]\!] \wedge [\![\mu : M]\!]$. A configuration $\mathtt{t}/\mu$ is *well-typed* if and only if there exist a type $\mathtt{T}$ and a store type M such that $dom(\mu) = dom(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$ is satisfiable. □

The type $\mathsf{ref}\,\mathtt{T}$ is the type of references (that is, memory locations) that store data of type $\mathtt{T}$. It must be *invariant* in its parameter, reflecting the fact that references may be *read* and *written*.

A store is a complex object: it may contain values that indirectly refer to each other via memory locations. In fact, it is a representation of the graph formed by objects and pointers in memory, which may contain cycles. We rely on store types to deal with such cycles. In the definition of well-typedness, the store type M imposes a constraint on the contents of the store—the value $\mu(m)$ must have type $M(m)$—but also plays the role of a hypothesis: by placing the constraint $[\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$ within the context $\mathsf{let\ ref}\, M\ \mathsf{in}\ []$, we give meaning to free occurrences of memory locations within $[\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$, and stipulate that it is valid to assume that m has type $M(m)$. In other words, we essentially view the store as a large, mutually recursive binding of locations to values. Since no satisfiable constraint may have a free program identifier (Lemma 1.3.31), every well-typed configuration must be closed. The context $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ []$ gives meaning to occurrences of constants within $[\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$.

We now define a relation between configurations that plays a key role in the statement of the subject reduction property. The point of subject reduction is to guarantee that well-typedness is preserved by reduction. However, such a simple statement is too weak to be amenable to inductive proof. Thus, for the purposes of the proof, we must be more specific. To begin, let us consider the simpler case of a pure semantics, that is, a semantics without stores. Then, we must state that if an expression $\mathtt{t}$ has type $\mathtt{T}$ under a certain constraint, then its reduct $\mathtt{t}'$ has type $\mathtt{T}$ under the same constraint. In terms of generated constraints, this statement becomes: $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t}' : \mathtt{T}]\!]$.

Let us now return to the general case, where a store is present. Then, the statement of well-typedness for a configuration $\mathtt{t}/\mu$ involves a store type M whose domain is that of μ. So, the statement of well-typedness for its reduct $\mathtt{t}'/\mu'$ must involve a store type M' whose domain is that of μ'—which is larger if allocation occurred. The types of existing memory locations must not change: we must request that M and M' agree on $dom(M)$, that is, M' must extend M. Furthermore, the types assigned to new memory locations in $dom(M') \setminus dom(M)$ might involve new type variables, that is, variables that do not appear free in M or $\mathtt{T}$. We must allow these variables to be hidden—that is, existentially quantified—otherwise the entailment assertion cannot hold. These considerations lead us to the following definition:

1.7.5 Definition: $\mathtt{t}/\mu \sqsubseteq \mathtt{t}'/\mu'$ holds if and only if, for every type $\mathtt{T}$ and for every store type M such that $dom(\mu) = dom(M)$, there exist a set of type variables $\bar{\mathtt{Y}}$ and a store type M' such that $\bar{\mathtt{Y}} \mathrel{\#} ftv(\mathtt{T}, M)$ and $ftv(M') \subseteq \bar{\mathtt{Y}} \cup ftv(M)$ and $dom(M') = dom(\mu')$ and M' extends M and

$$\begin{array}{c}\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathtt{T}/M]\!] \\ \Vdash \exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]\end{array}$$

The relation $\sqsubseteq$ is intended to express a connection between a configuration and its reduct. Thus, subject reduction may be stated as: $(\longrightarrow) \subseteq (\sqsubseteq)$, that is, $\sqsubseteq$ is indeed a conservative description of reduction. □

We have introduced an initial environment Γ_0 and used it in the definition of well-typedness, but we haven't yet ensured that the type schemes assigned to constants are an adequate description of their semantics. We now formulate two requirements that relate Γ_0 with $\xrightarrow{\delta}$. They are specializations of the subject reduction and progress properties to configurations that involve an application of a constant. They represent proof obligations that must be discharged when concrete definitions of $\mathcal{Q}$, $\xrightarrow{\delta}$, and Γ_0 are given.

1.7.6 Definition: We require (i) $(\xrightarrow{\delta}) \subseteq (\sqsubseteq)$; and (ii) if the configuration $\mathtt{c}\,\mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{c}\,\mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. □

The last point that remains to be settled before proving type soundness is the interaction between side effects and `let`-polymorphism. The following example illustrates the problem:

$$\mathtt{let}\ \mathtt{r} = \mathtt{ref}\ \lambda\mathtt{z}.\mathtt{z}\ \mathtt{in}\ \mathtt{let}\ \_ = (\mathtt{r} := \lambda\mathtt{z}.(\mathtt{z}\ \hat{+}\ \hat{1}))\ \mathtt{in}\ !\mathtt{r}\ \mathtt{true}$$

This expression reduces to $\mathtt{true}\ \hat{+}\ \hat{1}$, so it must not be well-typed. Yet, if natural type schemes are assigned to `ref`, !, and := (see Example 1.9.5), then it *is* well-typed with respect to the rules given so far, because `r` receives the polymorphic type scheme $\forall \mathtt{X}.\mathsf{ref}\,(\mathtt{X} \rightarrow \mathtt{X})$, which allows writing a function of type `int` $\rightarrow$ `int` into `r` and reading it back with type `bool` $\rightarrow$ `bool`. The problem is that `let`-polymorphism simulates a textual duplication of the `let`-bound expression `ref` λ`z.z`, while the semantics first reduces it to a value m, causing a new binding $m \mapsto \lambda\mathtt{z.z}$ to appear in the store, then duplicates the address m. The new store binding is not duplicated: both copies of m refer to the same memory cell. For this reason, generalization is unsound in this case, and must be restricted. Many authors have attempted to come up with a sound type system that accepts *all* pure programs and remains flexible enough in the presence of side effects (Tofte, 1988; Leroy, 1992). These proposals are often complex, which is why they have been abandoned in favor of an extremely simple *syntactic* restriction, known as the *value restriction* (Wright, 1995).

1.7.7 DEFINITION: A program satisfies the *value restriction* if and only if all subexpressions of the form $\mathtt{let\ z = t_1\ in\ t_2}$ are in fact of the form $\mathtt{let\ z = v_1\ in\ t_2}$. In the following, we assume that either all constants have pure semantics, or all programs satisfy the value restriction. □

Put slightly differently, the value restriction states that only values may be generalized. This eliminates the problem altogether, since duplicating values does not affect a program's semantics. Note that any program that does not satisfy the value restriction can be turned into one that does and has the same semantics: it suffices to change $\mathtt{let\ z = t_1\ in\ t_2}$ into $(\lambda\mathtt{z.t_2})\ \mathtt{t_1}$ when $\mathtt{t_1}$ is not a value. Of course, such a transformation may cause the program to become ill-typed. In other words, the value restriction causes some perfectly safe programs to be rejected. In particular, as stated above, it prevents generalizing applications of the form $\mathtt{c\ v_1\ \ldots\ v}_k$, where `c` is a destructor of arity k. This is excessive, because many destructors have pure semantics; only a few, such as `ref`, allocate new mutable storage. Furthermore, we use pure destructors to encode numerous language features (Section 1.9). Fortunately, it is easy to relax the restriction to allow generalizing not only values, but also a more general class of *nonexpansive* expressions, whose syntax guarantees that such expressions cannot allocate new mutable storage (that is, *expand* the domain of the store). The term *nonexpansive* was coined by Tofte (1988). Nonexpansive expressions may include applications of the form $\mathtt{c\ t_1\ \ldots\ t}_k$, where `c` is a pure destructor of arity k and $\mathtt{t}_1, \ldots, \mathtt{t}_k$ are nonexpansive. Experience shows that this slightly relaxed restriction is acceptable in practice. Some other improvements to the value restriction exist; see *e.g.* Exercise (Garrigue, 2002). Another frequent limitation of the value restriction are constructor functions, that is, functions that only build values, which are treated as ordinary functions and not as constructors, and their applications are not considered to be values. For instance, in the expression $\texttt{let f} = \texttt{c v in let z} = \texttt{f w in t}$ where $\texttt{c}$ is a constructor of arity 2, the partial application $\texttt{c v}$ bound to $\texttt{f}$ is a constructor function (of arity 1), but $\texttt{f w}$ is treated as a regular application and cannot be generalized. Technically, the effect of the (strict) value restriction is summarized by the following result.

1.7.8 Lemma: Under the value restriction, the production $\mathcal{E} ::= \texttt{let z} = \mathcal{E}\ \texttt{in t}$ may be suppressed from the grammar of evaluation contexts (Figure 1-1) without altering the operational semantics. □

We are done with definitions and requirements. We now come to the bulk of the type soundness proof.

1.7.9 Theorem [Subject reduction]: $(\longrightarrow) \subseteq (\sqsubseteq)$. □

Proof: Because $\longrightarrow$ and $\longrightarrow$ are the smallest relations that satisfy the rules of Figure 1-2, it suffices to prove that $\sqsubseteq$ satisfies these rules as well. We remark that if, for every type $\texttt{T}$, $[\![\texttt{t} : \texttt{T}]\!] \Vdash [\![\texttt{t}' : \texttt{T}]\!]$ holds, then $\texttt{t}/\mu \sqsubseteq \texttt{t}'/\mu$ holds. (Take $\bar{\texttt{Y}} = \varnothing$ and $M' = M$ and use the fact that entailment is a congruence to check that the conditions of Definition 1.7.5 are met.) We make use of this fact in cases R-Beta and R-Let below.

∘ *Case* R-Beta. We have

$$\begin{array}{lll} & [\![(\lambda \texttt{z.t})\ \texttt{v} : \texttt{T}]\!] & \\ \equiv & \exists \texttt{X}.([\![\lambda \texttt{z.t} : \texttt{X} \rightarrow \texttt{T}]\!] \wedge [\![\texttt{v} : \texttt{X}]\!]) & \textbf{(1)} \\ \equiv & \exists \texttt{X}.(\mathsf{let}\ \texttt{z} : \texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] \wedge [\![\texttt{v} : \texttt{X}]\!]) & \textbf{(2)} \\ \equiv & \exists \texttt{X}.\mathsf{let}\ \texttt{z} : \forall \varnothing[[\![\texttt{v} : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] & \textbf{(3)} \\ \Vdash & [\![[\texttt{z} \mapsto \texttt{v}]\texttt{t} : \texttt{T}]\!] & \textbf{(4)} \end{array}$$

where (1) is by definition of constraint generation; (2) is by Lemma 1.6.5; (3) is by C-LetAnd; (4) is by Lemma 1.7.2 and C-Ex*.

∘ *Case* R-Let. We have

$$\begin{array}{lll} & [\![\texttt{let z} = \texttt{v in t} : \texttt{T}]\!] & \\ = & \mathsf{let}\ \texttt{z} : \forall \texttt{X}[[\![\texttt{v} : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t} : \texttt{T}]\!] & \textbf{(1)} \\ \Vdash & [\![[\texttt{z} \mapsto \texttt{v}]\texttt{t} : \texttt{T}]\!] & \textbf{(2)} \end{array}$$

where (1) is by definition of constraint generation and (2) is by Lemma 1.7.2.

∘ *Case* R-Delta. This case is exactly requirement (i) in Definition 1.7.6.

∘ *Case* R-Extend. Our hypotheses are $\texttt{t}/\mu \sqsubseteq \texttt{t}'/\mu'$ **(1)** and $\mathit{dom}(\mu'') \# \mathit{dom}(\mu')$ **(2)** and $\mathit{range}(\mu'') \# \mathit{dom}(\mu' \setminus \mu)$ **(3)**. Because $\mathit{dom}(\mu)$ must be a subset of $\mathit{dom}(\mu')$, it is also disjoint with $\mathit{dom}(\mu'')$. Our goal is $\texttt{t}/\mu\mu'' \sqsubseteq \texttt{t}'/\mu'\mu''$ **(4)**. Thus, let us introduce a type $\texttt{T}$ and a store type of domain

$dom(\mu\mu'')$, or (equivalently) two store types M and M'' whose domains are respectively $dom(\mu)$ and $dom(\mu'')$. By (1), there exist type variables $\bar{\mathtt{Y}}$ and a store type M' such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}, M)$ **(5)** and $\mathit{ftv}(M') \subseteq \bar{\mathtt{Y}} \cup \mathit{ftv}(M)$ and $dom(M') = dom(\mu')$ and M' extends M **(6)** and $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathtt{T}/M]\!] \Vdash \exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]$. We may further require, *w.l.o.g.*, $\bar{\mathtt{Y}} \# \mathit{ftv}(M'')$ **(7)**. Let us now add the conjunct $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mu'' : M'']\!]$ to each side of this entailment assertion. On the left-hand side, by C-InAnd and by Definition 1.7.4, we obtain $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}/\mu\mu'' : \mathtt{T}/MM'']\!]$ **(8)**. On the right-hand side, by (5), (7), C-ExAnd, and C-InAnd, we obtain $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{let}\ \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!] \wedge \mathsf{let}\ \mathsf{ref}\, M\ \mathsf{in}\ [\![\mu'' : M'']\!])$ **(9)**. Now, recall that M' extends M (6) and, furthermore, (3) implies $\mathit{fpi}([\![\mu'' : M'']\!]) \# \mathit{dpi}(M' \setminus M)$ **(10)**. By (10), C-InAnd*, and C-InAnd, (9) is equivalent to $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ ([\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!] \wedge [\![\mu'' : M'']\!])$, that is, $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu'\mu'' : \mathtt{T}/M'M'']\!]$ **(11)**. Thus, we have established that (8) entails (11). Let us now place this entailment assertion within the constraint context $\mathsf{let}\ \mathsf{ref}\, M''\ \mathsf{in}\ []$. On the left-hand side, because $\mathit{fpi}(\Gamma_0, M, M'') = \varnothing$ and $\mathit{dpi}(M'') \cap \mathit{dpi}(\Gamma_0, M) \subseteq dom(\mu'') \cap (\mathcal{Q} \cup dom(\mu)) = \varnothing$, C-LetLet applies, yielding $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, MM''\ \mathsf{in}\ [\![\mathtt{t}/\mu\mu'' : \mathtt{T}/MM'']\!]$ **(12)**. On the right-hand side, by (7), C-InEx, and by analogous reasoning, we obtain $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'M''\ \mathsf{in}\ [\![\mathtt{t}'/\mu'\mu'' : \mathtt{T}/M'M'']\!]$ **(13)**. Thus, (12) entails (13). Given (5), (7), given $\mathit{ftv}(M'M'') \subseteq \bar{\mathtt{Y}} \cup \mathit{ftv}(MM'')$, and given that $M'M''$ extends MM'', this establishes the goal (4).

∘ *Case* R-Context. The hypothesis is $\mathtt{t}/\mu \sqsubseteq \mathtt{t}'/\mu'$. The goal is $\mathcal{E}[\mathtt{t}]/\mu \sqsubseteq \mathcal{E}[\mathtt{t}']/\mu'$. Because $\longrightarrow$ relates closed configurations only, we may assume that the configuration $\mathcal{E}[\mathtt{t}]/\mu$ is closed, so the memory locations that appear free within $\mathcal{E}$ are members of $dom(\mu)$. Let us now reason by induction on the structure of $\mathcal{E}$.

Subcase $\mathcal{E} = []$. The hypothesis and the goal coincide.

Subcase $\mathcal{E} = \mathcal{E}_1\ \mathtt{t}_1$. The induction hypothesis is $\mathcal{E}_1[\mathtt{t}]/\mu \sqsubseteq \mathcal{E}_1[\mathtt{t}']/\mu'$ **(1)**. Let us introduce a type $\mathtt{T}$ and a store type M such that $dom(M) = dom(\mu)$. Consider the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathcal{E}[\mathtt{t}]/\mu : \mathtt{T}/M]\!]$ **(2)**. By definition of constraint generation, C-ExAnd, C-InEx, and C-InAnd, it is equivalent to

$$\exists\mathtt{X}.(\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}]/\mu : \mathtt{X} \to \mathtt{T}/M]\!] \wedge \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{X}]\!])\ (\mathbf{3})$$

where $\mathtt{X} \notin \mathit{ftv}(\mathtt{T}, M)$ **(4)**. By (1), there exist type variables $\bar{\mathtt{Y}}$ and a store type M' such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{X}, \mathtt{T}, M)$ **(5)** and $\mathit{ftv}(M') \subseteq \bar{\mathtt{Y}} \cup \mathit{ftv}(M)$ **(6)** and $dom(M') = dom(\mu')$ and M' extends M and (3) entails

$$\exists\mathtt{X}.(\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathcal{E}_1[\mathtt{t}']/\mu' : \mathtt{X} \to \mathtt{T}/M']\!] \wedge \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{X}]\!])\ (\mathbf{7}).$$

We pointed out earlier that the memory locations that appear free in $\mathtt{t}_1$ are members of $dom(M)$, which implies $\mathsf{let}\ \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{X}]\!] \equiv \mathsf{let}\ \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}_1 :$

$\texttt{X}]\!]$ **(8)**. By (5), C-ExAnd, (8), C-InAnd, and by definition of constraint generation, we find that (7) is equivalent to

$$\exists\texttt{X}\bar{\texttt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ ([\![\mathcal{E}_1[\texttt{t}'] : \texttt{X} \to \texttt{T}]\!] \wedge [\![\texttt{t}_1 : \texttt{X}]\!] \wedge [\![\mu' : M']\!])\ (\mathbf{9}).$$

(4), (5) and (6) imply $\texttt{X} \notin \mathit{ftv}(M')$. Thus, by C-InEx and C-ExAnd, (9) may be written

$$\exists\bar{\texttt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ (\exists\texttt{X}.([\![\mathcal{E}_1[\texttt{t}'] : \texttt{X} \to \texttt{T}]\!] \wedge [\![\texttt{t}_1 : \texttt{X}]\!]) \wedge [\![\mu' : M']\!]),$$

which, by definition of constraint generation, is

$$\exists\bar{\texttt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}[\texttt{t}']/\mu' : \texttt{T}/M']\!]\ (\mathbf{10}).$$

Thus, we have proved that (2) entails (10). By Definition 1.7.5, this establishes $\mathcal{E}[\texttt{t}]/\mu \sqsubseteq \mathcal{E}[\texttt{t}']/\mu'$.

Subcase $\mathcal{E} = \texttt{v}\ \mathcal{E}_1$. Analogous to the previous subcase.

Subcase $\mathcal{E} = \texttt{let z =}\ \mathcal{E}_1\ \texttt{in t}_1$. The induction hypothesis is $\mathcal{E}_1[\texttt{t}]/\mu \sqsubseteq \mathcal{E}_1[\texttt{t}']/\mu'$ **(1)**. This subcase is particularly interesting, because it is where `let`-polymorphism and side effects interact. In the previous two subcases, we relied on the fact that the $\exists\bar{\texttt{Y}}$ quantifier, which hides the types of the memory cells created by the reduction step, *commutes* with the connectives $\exists$ and $\wedge$ introduced by application contexts. However, it does not in general (left-)commute with the let connective (Example 1.3.28). Fortunately, under the value restriction, this subcase *never arises* (Lemma 1.7.8). By Definition 1.7.7, this subcase may arise only if all constants have pure semantics, which implies $\mu = \mu' = \varnothing$. Then, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}[\texttt{t}] : \texttt{T}]\!] & \\
= & \mathsf{let}\ \Gamma_0; \texttt{z} : \forall\texttt{X}[[\![\mathcal{E}_1[\texttt{t}] : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & (\mathbf{2}) \\
\equiv & \mathsf{let}\ \Gamma_0; \texttt{z} : \forall\texttt{X}[\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}_1[\texttt{t}] : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & (\mathbf{3}) \\
\Vdash & \mathsf{let}\ \Gamma_0; \texttt{z} : \forall\texttt{X}[\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}_1[\texttt{t}'] : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & (\mathbf{4}) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathcal{E}[\texttt{t}'] : \texttt{T}]\!] & (\mathbf{5})
\end{array}$$

where (2) is by definition of constraint generation; (3) follows from $\mathit{ftv}(\Gamma_0) = \mathit{fpi}(\Gamma_0) = \varnothing$ and C-LetDup; (4) follows from (1), specialized to the case of a pure semantics; and (5) is obtained by performing these steps in reverse. □

1.7.10 Exercise [Recommended, ★★★]: Try to carry out the last subcase of the above proof in the case of an impure semantics and in the absence of the value restriction. Find out why it fails. Show that it succeeds if $\bar{\texttt{Y}}$ is assumed to be empty. Use this fact to prove that generalization is still safe when restricted to *nonexpansive* expressions, provided (i) evaluating a nonexpansive expression cannot cause new memory cells to be allocated, (ii) nonexpansive expressions are stable by substitution of values for variables, and (iii) nonexpansive expressions are preserved by reduction. □

Subject reduction ensures that well-typedness is preserved by reduction.

1.7.11 LEMMA: Let $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$. If $\mathtt{t}/\mu$ is well-typed, then so is $\mathtt{t}'/\mu'$. □

Proof: Assume $\mathtt{t}/\mu \longrightarrow \mathtt{t}'/\mu'$ **(1)** and $\mathtt{t}/\mu$ is well-typed **(2)**. By (2) and Definition 1.7.4, there exist a type $\mathtt{T}$ and a store type M such that $dom(\mu) = dom(M)$ and the constraint $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{t}/\mu : \mathtt{T}/M]\!]$ **(3)** is satisfiable. By Theorem 1.7.9 and Definition 1.7.5, (1) implies that there exist a set of type variables $\bar{\mathtt{Y}}$ and a store type M' such that $dom(M') = dom(\mu')$ **(4)** and the constraint (3) entails $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]$ **(5)**. Because (3) is satisfiable, so is (5), which implies that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![\mathtt{t}'/\mu' : \mathtt{T}/M']\!]$ is satisfiable **(6)**. By (4) and (6) and Definition 1.7.4, $\mathtt{t}'/\mu'$ is well-typed. □

Let us now establish the progress property.

1.7.12 LEMMA: If $\mathtt{t}_1\ \mathtt{t}_2$ is well-typed, then $\mathtt{t}_1/\mu$ and $\mathtt{t}_2/\mu$ are well-typed. If $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2/\mu$ is well-typed, then $\mathtt{t}_1/\mu$ is well-typed. □

1.7.13 THEOREM [PROGRESS]: If $\mathtt{t}/\mu$ is well-typed, then either it is reducible, or $\mathtt{t}$ is a value. □

Proof: The proof is by induction on the structure of $\mathtt{t}$.

∘ *Case* $\mathtt{t} = \mathtt{z}$. Well-typed configurations are closed: this case cannot occur.

∘ *Case* $\mathtt{t} = m$. $\mathtt{t}$ is a value.

∘ *Case* $\mathtt{t} = \mathtt{c}$. By requirement (ii) of Definition 1.7.6.

∘ *Case* $\mathtt{t} = \lambda\mathtt{z}.\mathtt{t}_1$. $\mathtt{t}$ is a value.

∘ *Case* $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$. By Lemma 1.7.12, $\mathtt{t}_1/\mu$ is well-typed. By the induction hypothesis, either it is reducible, or $\mathtt{t}_1$ is a value. If the former, by R-CONTEXT and because every context of the form $\mathcal{E}\ \mathtt{t}_2$ is an evaluation context, the configuration $\mathtt{t}/\mu$ is reducible as well. Thus, let us assume $\mathtt{t}_1$ is a value. By Lemma 1.7.12, $\mathtt{t}_2/\mu$ is well-typed. By the induction hypothesis, either it is reducible, or $\mathtt{t}_2$ is a value. If the former, by R-CONTEXT and because every context of the form $\mathtt{t}_1\ \mathcal{E}$—where $\mathtt{t}_1$ is a value—is an evaluation context, the configuration $\mathtt{t}/\mu$ is reducible as well. Thus, let us assume $\mathtt{t}_2$ is a value. Let us now reason by cases on the structure of $\mathtt{t}_1$.

Subcase $\mathtt{t}_1 = \mathtt{z}$. Again, this subcase cannot occur.

Subcase $\mathtt{t}_1 = m$. Because $\mathtt{t}/\mu$ is well-typed, a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ (\exists\mathtt{X}.(m \preceq \mathtt{X} \to \mathtt{T} \wedge [\![\mathtt{t}_2 : \mathtt{X}]\!]) \wedge [\![\mu : M]\!])$ must be satisfiable. This implies that m is a member of $dom(M)$ and that the constraint

$\mathsf{ref}\, M(m) \leq \mathtt{X} \rightarrow \mathtt{T}$ is satisfiable. Because the type constructors ref and $\rightarrow$ are incompatible, this is a contradiction. So, this subcase cannot occur.

Subcase $\mathtt{t}_1 = \lambda\mathtt{z}.\mathtt{t}'_1$. By R-Beta, $\mathtt{t}/\mu$ is reducible.

Subcase $\mathtt{t}_1 = \mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$. Then, $\mathtt{t}$ is of the form $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_{k+1}$. The result follows by requirement (ii) of Definition 1.7.6.

∘ *Case* $\mathtt{t} = \mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$. By Lemma 1.7.12, $\mathtt{t}_1/\mu$ is well-typed. By the induction hypothesis, either $\mathtt{t}_1/\mu$ is reducible, or $\mathtt{t}_1$ is a value. If the former, by R-Context and because every context of the form $\mathtt{let}\ \mathtt{z} = \mathcal{E}\ \mathtt{in}\ \mathtt{t}_2$ is an evaluation context, the configuration $\mathtt{t}/\mu$ is reducible as well. If the latter, then $\mathtt{t}/\mu$ is reducible by R-Let. □

We may now conclude:

1.7.14 Theorem [Type Soundness]: Well-typed source programs do not go wrong. □

Proof: We say that a source program $\mathtt{t}$ is well-typed if and only if the configuration $\mathtt{t}/\varnothing$ is well-typed, that is, if and only if $\exists\mathtt{X}.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{t} : \mathtt{X}]\!] \equiv \mathsf{true}$ holds. By Lemma 1.7.11, all reducts of $\mathtt{t}/\varnothing$ are well-typed. By Theorem 1.7.13, none is stuck. □

Let us recall that this result holds only if the requirements of Definition 1.7.6 are met. In other words, some proof obligations remain to be discharged when concrete definitions of $\mathcal{Q}$, $\xrightarrow{\delta}$, and Γ_0 are given. This is illustrated by several examples in the next section.

1.8 Constraint solving

We have introduced a parameterized constraint language, given equivalence laws that describe the interaction between its logical connectives, and exploited them to prove theorems about type inference and type soundness, which are valid independently of the nature of primitive constraints—the so-called predicate applications. However, there would be little point in proposing a parameterized constraint solver, because much of the difficulty of designing an efficient constraint solver precisely lies in the treatment of primitive constraints and in its interaction with `let`-polymorphism. For this reason, in this section, we focus on constraint solving in the setting of an *equality-only free tree model.* Thus, the constraint solver developed here allows performing type inference for HM(=) (that is, for Damas and Milner's type system) and for its extension with recursive types. Of course, some of its mechanisms may be useful in other settings. Other constraint solvers used in program analysis or type inference are described *e.g.* in (Aiken and Wimmers, 1992; Niehren, Müller, and Podelski, 1997; Fähndrich, 1999; Melski and Reps, 2000; Müller, Niehren, and Treinen, 2001; Pottier, 2001b; Nielson, Nielson, and Seidl, 2002; McAllester, 2002, 2003).

We begin with a rule-based presentation of a standard, efficient first-order unification algorithm. This yields a constraint solver for a subset of the constraint language, deprived of type scheme introduction and instantiation forms. On top of it, we build a full constraint solver, which corresponds to the code that accompanies this chapter.

Unification

Unification is the process of solving equations between terms. We now present a unification algorithm due to Huet (1976) as a (nondeterministic) system of constraint rewriting rules. The specification is almost the same in the case of *finite* and *regular* tree models: only one rule, which implements the *occurs check*, must be removed in the latter case. In other words, the algorithm works with *possibly cyclic* terms, and does not rely in an essential way on the occurs check. In order to accurately reflect the behavior of the actual algorithm, which relies on a *union-find* data structure (Tarjan, 1975), we modify the syntax of constraints by replacing equations with *multi-equations*. A multi-equation is an equation that involves an arbitrary number of types, as opposed to exactly two.

1.8.1 Definition: Let there be, for every kind κ and for every $n \geq 1$, a predicate $=^n_\kappa$, of signature $\kappa^n \Rightarrow \cdot$, whose interpretation is (n-ary) equality. The predicate constraint $=^n_\kappa\ \mathtt{T}_1 \ldots \mathtt{T}_n$ is written $\mathtt{T}_1 = \ldots = \mathtt{T}_n$, and called a *multi-equation*. We consider the constraint true as a multi-equation of length 0. In the following, we identify multi-equations up to permutations of their members, so a multi-equation ϵ of kind κ may be viewed as a finite *multi-set* of types of kind κ. We write $\epsilon = \epsilon'$ for the multi-equation obtained by concatenating ϵ and ϵ'. □

Thus, we are interested in the following subset of the constraint language:

$$U ::= \mathsf{true} \mid \mathsf{false} \mid \epsilon \mid U \wedge U \mid \exists \bar{\mathtt{X}}.U$$

Equations are replaced with multi-equations; no other predicates are available. Type scheme introduction and instantiation forms are absent.

1.8.2 Definition: A multi-equation is *standard* if and only if its variable members are distinct and it has at most one nonvariable member. A constraint U is *standard* if and only if every multi-equation inside U is standard and every variable that occurs (free or bound) in U is a member of at most one multi-equation inside U. □

A union-find algorithm maintains equivalence classes (that is, disjoint sets) of variables, and associates, with each class, a *descriptor*, which in our case is either absent or a nonvariable term. Thus, a *standard* constraint represents a state of the union-find algorithm. A constraint that is *not* standard may be viewed as a superposition of a state of the union-find algorithm, on the one hand, and of control information, on the other hand. For instance, a multi-equation of the form $\epsilon = \mathtt{T}_1 = \mathtt{T}_2$, where $\mathtt{T}_1$ and $\mathtt{T}_2$ are nonvariable terms, may be viewed, roughly speaking, as the equivalence class $\epsilon = \mathtt{T}_1$, together with a pending request to solve $\mathtt{T}_1 = \mathtt{T}_2$ and to update the class's descriptor accordingly. Because multi-equations encode both state and control, our specification of unification is rather high-level. It would be possible to give a lower-level description, where state (standard conjunctions of multi-equations) and control (pending binary equations) are distinguished.

1.8.3 Definition: Let U be a conjunction of multi-equations. $\mathtt{Y}$ is *dominated* by $\mathtt{X}$ with respect to U (written: $\mathtt{Y} \prec_U \mathtt{X}$) if and only if U contains a conjunct of the form $\mathtt{X} = F\,\vec{\mathtt{T}} = \epsilon$, where $\mathtt{Y} \in ftv(\bar{\mathtt{T}})$. U is *cyclic* if and only if the graph of $\prec_U$ exhibits a cycle. □

The specification of the unification algorithm consists of a set of constraint rewriting rules, given in Figure 1-11. Rewriting is performed modulo α-conversion, modulo permutations of the members of a multi-equation, modulo commutativity and associativity of conjunction, and under an arbitrary context. The specification is nondeterministic: several rule instances may be simultaneously applicable.

S-ExAnd is a directed version of C-ExAnd, whose effect is to float up all existential quantifiers. In the process, all multi-equations become part of a single conjunction, possibly causing rules whose left-hand side is a conjunction of multi-equations, namely S-Fuse and S-Cycle, to become applicable. S-Fuse identifies two multi-equations that share a common variable $\mathtt{X}$, and fuses them. The new multi-equation is not necessarily standard, even if the two original multi-equations were. Indeed, it may have repeated variables or contain two nonvariable terms. The purpose of the next few rules, whose left-hand side consists of a single multi-equation, is to deal with these situations. S-Stutter eliminates redundant variables. It only deals with variables, as opposed to terms of arbitrary size, so as to have constant time cost. The comparison of nonvariable terms is implemented by S-Decompose and S-Clash. S-Decompose decomposes an equation between two terms whose head symbols match. It produces a conjunction of equations between their subterms, namely $\vec{\mathtt{X}} = \vec{\mathtt{T}}$. Only one of the two terms remains in the original multi-equation, which may thus become standard. The terms $\vec{\mathtt{X}}$ are copied—there are two occurrences of $\vec{\mathtt{X}}$ on the right-hand side. For this reason, we

$$
\begin{array}{rcll}
(\exists\bar{\mathtt{X}}.U_1) \wedge U_2 & \rightarrow & \exists\bar{\mathtt{X}}.(U_1 \wedge U_2) & \text{(S-ExAnd)} \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(U_2) & \\
\mathtt{X} = \epsilon \wedge \mathtt{X} = \epsilon' & \rightarrow & \mathtt{X} = \epsilon = \epsilon' & \text{(S-Fuse)} \\
\mathtt{X} = \mathtt{X} = \epsilon & \rightarrow & \mathtt{X} = \epsilon & \text{(S-Stutter)} \\
F\,\vec{\mathtt{X}} = F\,\vec{\mathtt{T}} = \epsilon & \rightarrow & \vec{\mathtt{X}} = \vec{\mathtt{T}} \wedge F\,\vec{\mathtt{X}} = \epsilon & \text{(S-Decompose)} \\
F\,\mathtt{T}_1\,\ldots\,\mathtt{T}_i\,\ldots\,\mathtt{T}_n = \epsilon & \rightarrow & \exists\mathtt{X}.(\mathtt{X} = \mathtt{T}_i \wedge F\,\mathtt{T}_1\,\ldots\,\mathtt{X}\,\ldots\,\mathtt{T}_n = \epsilon) & \text{(S-Name-1)} \\
 & & \text{if } \mathtt{T}_i \notin \mathcal{V} \wedge \mathtt{X} \notin \mathit{ftv}(\mathtt{T}_1, \ldots, \mathtt{T}_n, \epsilon) & \\
F\,\vec{\mathtt{T}} = F'\,\vec{\mathtt{T}}' = \epsilon & \rightarrow & \mathsf{false} & \text{(S-Clash)} \\
 & & \text{if } F \neq F' & \\
\mathtt{T} & \rightarrow & \mathsf{true} & \text{(S-Single)} \\
 & & \text{if } \mathtt{T} \notin \mathcal{V} & \\
U \wedge \mathsf{true} & \rightarrow & U & \text{(S-True)} \\
U & \rightarrow & \mathsf{false} & \text{(S-Cycle)} \\
 & & \text{if the model is syntactic and } U \text{ is cyclic} & \\
\mathcal{U}[\mathsf{false}] & \rightarrow & \mathsf{false} & \text{(S-Fail)} \\
 & & \text{if } \mathcal{U} \neq [] &
\end{array}
$$

Figure 1-11: Unification

require them to be type variables, as opposed to terms of arbitrary size. (We slightly abuse notation by using $\vec{\mathtt{X}}$ to denote a vector of type variables whose elements are *not* necessarily distinct.) By doing so, we allow explicitly reasoning about *sharing*: since a variable represents a pointer to an equivalence class, we explicitly specify that only *pointers*, not whole terms, are copied. As a result of this decision, S-Decompose is not applicable when both terms at hand have a nonvariable subterm. S-Name-1 remedies this problem by introducing a fresh variable that stands for one such subterm. When repeatedly applied, S-Name-1 yields a unification problem composed of so-called *small terms* only—that is, where sharing has been made fully explicit. S-Clash complements S-Decompose by dealing with the case where two terms with different head symbols are equated; in a free tree model, such an equation is false, so failure is signaled. S-Single and S-True suppress multi-equations of size 1 and 0, respectively, which are tautologies. S-Single is restricted to nonvariable terms so as not to break the property that every variable is a member of exactly one multi-equation (Definition 1.8.2). S-Cycle is the occurs check: that is, it signals failure if the constraint is cyclic. It is applicable only in the case of syntactic unification, that is, when ground types are finite trees. It is a global check: its left-hand side is an entire conjunction of multi-equations. S-Fail propagates failure; $\mathcal{U}$ ranges over unification constraint contexts.

The constraint rewriting system in Figure 1-11 enjoys the following properties. First, rewriting is strongly normalizing, so the rules define a (nondeterministic) algorithm. Second, rewriting is meaning-preserving. Third, every normal form is either false or of the form $\exists\bar{\mathtt{X}}.U$, where U is satisfiable. The latter two properties indicate that the algorithm is indeed a constraint solver.

1.8.4 Lemma: The rewriting system $\rightarrow$ is strongly normalizing. $\square$

1.8.5 Lemma: $U_1 \rightarrow U_2$ implies $U_1 \equiv U_2$. $\square$

1.8.6 Lemma: Every normal form is either false or of the form $\mathcal{X}[U]$, where $\mathcal{X}$ is an existential constraint context, U is a standard conjunction of multi-equations and, if the model is syntactic, U is acyclic. These conditions imply that U is satisfiable. $\square$

A constraint solver

On top of the unification algorithm, we now define a constraint solver. Its specification is independent of the rules and strategy employed by the unification algorithm. However, the structure of the unification algorithm's normal forms, as well as the logical properties of multi-equations, are exploited when performing generalization, that is, when creating and simplifying type schemes. Like the unification algorithm, the constraint solver is specified in terms of a *reduction system*. However, the objects that are subject to rewriting are not just constraints: they have more complex structure. Working with such richer *states* allows distinguishing the solver's external language—namely, the full constraint language, which is used to express the problem that one wishes to solve—and an internal language, introduced below, which is used to describe the solver's private data structures. In the following, C and D range over *external* constraints, that is, constraints that were part of the solver's input. External constraints are to be viewed as abstract syntax trees, subject to no implicit laws other than α-conversion. As a simplifying assumption, we require external constraints not to contain any occurrence of false—otherwise the problem at hand is clearly false. *Internal* data structures include unification constraints U, as previously studied, and *stacks*. Stacks form a subset of constraint contexts, defined on page 24. Their syntax is as follows:

$$S ::= [] \mid S[[] \wedge C] \mid S[\exists\bar{\mathtt{X}}.[]] \mid S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C] \mid S[\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ []]$$

In the second and fourth productions, C is an external constraint. In the last production, we require σ to be of the form $\forall\bar{\mathtt{X}}[U].\mathtt{X}$, and we demand $\exists\sigma \equiv \mathsf{true}$. A stack may be viewed as a list of *frames*. Frames may be added and deleted at the inner end of a stack, that is, near the hole of the constraint context that it represents. We refer to the four kinds of frames as *conjunction*, *existential*, let, and *environment* frames, respectively. A *state* of the constraint solver is a triple $S;U;C$, where S is a stack, U is a unification constraint, and C is an external constraint. The state $S;U;C$ is to be understood as a representation of the constraint $S[U \wedge C]$. The notion of α-equivalence between states is defined accordingly. In particular, one may rename type variables in $dtv(S)$, provided U and C are renamed as well. In short, the three components of a state play the following roles. C is an external constraint that the solver intends to examine next. U is the internal state of the underlying unification algorithm: one might think of it as the knowledge that has been obtained so far. S tells where the type variables that occur free in U and C are bound, associates type schemes with the program variables that occur free in C, and records what should be done after C is solved. The solver's initial state is usually of the form $[];\mathsf{true};C$, where C is the external constraint that one wishes to solve—that is, whose satisfiability one wishes to determine. For simplicity, we make the (unessential) assumption that states have no free type variables.

The solver consists of a (nondeterministic) state rewriting system, given in Figure 1-12. Rewriting is performed modulo α-conversion. S-Unify makes the unification algorithm a component of the constraint solver, and allows the current unification problem U to be solved at any time. Rules S-Ex-1 to S-Ex-4 float existential quantifiers out of the unification problem into the stack, and through the stack up to the nearest enclosing let frame, if there is any, or to the outermost level, otherwise. Their side-conditions prevent capture of type variables, and may always be satisfied by suitable α-conversion of the left-hand state. If $S;U;C$ is a normal form with respect to the above five rules, then every type variable in $dtv(S)$ is either universally quantified at a let frame, or existentially bound at the outermost level. (Recall that, by assumption, states have no free type variables.) In other words, provided these rules are applied in an eager fashion, *there is no need for existential frames to appear in the machine representation of stacks*. Instead, it suffices to maintain, at every let frame and at the outermost level, a list of the type variables that are bound at this point; and, conversely, to annotate every type variable in $dtv(S)$ with an integer *rank*, which allows telling, in constant time, where the variable is bound: type variables of rank 0 are bound at the outermost level, and type variables of rank $k \geq 1$ are bound at the k^{th} let frame down in the stack S. The code that accompanies this chapter adopts this convention. Ranks were

$$
\begin{array}{rcll}
S; U; C & \rightarrow & S; U'; C & (\textsc{S-Unify}) \\
 & & \text{if } U \rightarrow U' & \\
S; \exists\bar{\mathtt{X}}.U; C & \rightarrow & S[\exists\bar{\mathtt{X}}.[]]; U; C & (\textsc{S-Ex-1}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(C) & \\
S[(\exists\bar{\mathtt{X}}.[]) \wedge C] & \rightarrow & S[\exists\bar{\mathtt{X}}.([] \wedge C)] & (\textsc{S-Ex-2}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(C) & \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[\exists\bar{\mathtt{Y}}.[]].\mathtt{T}\ \mathsf{in}\ C] & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[[]].\mathtt{T}\ \mathsf{in}\ C] & (\textsc{S-Ex-3}) \\
 & & \text{if } \bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}) & \\
S[\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.[]] & \rightarrow & S[\exists\bar{\mathtt{X}}.\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ []] & (\textsc{S-Ex-4}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(\sigma) & \\
S; U; \mathtt{T}_1 = \mathtt{T}_2 & \rightarrow & S; U \wedge \mathtt{T}_1 = \mathtt{T}_2; \mathsf{true} & (\textsc{S-Solve-Eq}) \\
S; U; \mathtt{x} \preceq \mathtt{T} & \rightarrow & S; U; S(\mathtt{x}) \preceq \mathtt{T} & (\textsc{S-Solve-Id}) \\
S; U; C_1 \wedge C_2 & \rightarrow & S[[] \wedge C_2]; U; C_1 & (\textsc{S-Solve-And}) \\
S; U; \exists\bar{\mathtt{X}}.C & \rightarrow & S[\exists\bar{\mathtt{X}}.[]]; U; C & (\textsc{S-Solve-Ex}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(U) & \\
S; U; \mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[D].\mathtt{T}\ \mathsf{in}\ C & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C]; U; D & (\textsc{S-Solve-Let}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(U) & \\
S[[] \wedge C]; U; \mathsf{true} & \rightarrow & S; U; C & (\textsc{S-Pop-And}) \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C]; U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\mathtt{X}[[]].\mathtt{X}\ \mathsf{in}\ C]; & \\
 & & U \wedge \mathtt{X} = \mathtt{T}; \mathsf{true} & (\textsc{S-Name-2}) \\
 & & \text{if } \mathtt{X} \notin \mathit{ftv}(U, \mathtt{T}) \wedge \mathtt{T} \notin \mathcal{V} & \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\mathtt{Y}[[]].\mathtt{X}\ \mathsf{in}\ C]; \mathtt{Y} = \mathtt{Z} = \epsilon \wedge U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\mathtt{Y}[[]].\theta(\mathtt{X})\ \mathsf{in}\ C]; & \\
 & & \mathtt{Y} \wedge \mathtt{Z} = \theta(\epsilon) \wedge \theta(U); \mathsf{true} & (\textsc{S-Compress}) \\
 & & \text{if } \mathtt{Y} \neq \mathtt{Z} \wedge \theta = [\mathtt{Y} \mapsto \mathtt{Z}] & \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\mathtt{Y}[[]].\mathtt{X}\ \mathsf{in}\ C]; \mathtt{Y} = \epsilon \wedge U; \mathsf{true} & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{X}\ \mathsf{in}\ C]; \epsilon \wedge U; \mathsf{true} & (\textsc{S-UnName}) \\
 & & \text{if } \mathtt{Y} \notin \mathtt{X} \cup \mathit{ftv}(\epsilon, U) & \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[[]].\mathtt{X}\ \mathsf{in}\ C]; U; \mathsf{true} & \rightarrow & S[\exists\bar{\mathtt{Y}}.\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{X}\ \mathsf{in}\ C]; U; \mathsf{true} & (\textsc{S-LetAll}) \\
 & & \text{if } \bar{\mathtt{Y}} \# \mathit{ftv}(C) \wedge \exists\bar{\mathtt{X}}.U \text{ determines } \bar{\mathtt{Y}} & \\
S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{X}\ \mathsf{in}\ C]; U_1 \wedge U_2; \mathsf{true} & \rightarrow & S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[U_2].\mathtt{X}\ \mathsf{in}\ []]; U_1; C & (\textsc{S-Pop-Let}) \\
 & & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(U_1) \wedge \exists\bar{\mathtt{X}}.U_2 \equiv \mathsf{true} & \\
S[\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ []]; U; \mathsf{true} & \rightarrow & S; U; \mathsf{true} & (\textsc{S-Pop-Env})
\end{array}
$$

Figure 1-12: A constraint solver

initially described in (Rémy, 1992a), and also appear in (McAllester, 2003).

Rules S-Solve-Eq to S-Solve-Let encode an analysis of the structure of the third component of the current state. There is one rule for each possible case, except false, which by assumption cannot arise, and true, which is dealt with further on. S-Solve-Eq discovers an equation and makes it available to the unification algorithm. S-Solve-Id discovers an instantiation constraint $\mathtt{x} \preceq \mathtt{T}$ and replaces it with $\sigma \preceq \mathtt{T}$, where the type scheme $\sigma = S(\mathtt{x})$ is the type scheme carried by the nearest environment frame that defines $\mathtt{x}$ in the stack S. It is defined as follows:

$$\begin{array}{rcll} S[[] \wedge C](\mathtt{x}) & = & S(\mathtt{x}) & \\ S[\exists\bar{\mathtt{X}}.[]](\mathtt{x}) & = & S(\mathtt{x}) & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(S(\mathtt{x})) \\ S[\mathsf{let}\ \mathtt{y} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C](\mathtt{x}) & = & S(\mathtt{x}) & \text{if } \bar{\mathtt{X}} \# \mathit{ftv}(S(\mathtt{x})) \\ S[\mathsf{let}\ \mathtt{y} : \sigma\ \mathsf{in}\ []](\mathtt{x}) & = & S(\mathtt{x}) & \text{if } \mathtt{x} \neq \mathtt{y} \\ S[\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ []](\mathtt{x}) & = & \sigma & \end{array}$$

If $\mathtt{x} \in \mathit{dpi}(S)$ does not hold, then $S(\mathtt{x})$ is undefined and the rule is not applicable. If it does hold, then the rule may always be made applicable by suitable α-conversion of the left-hand state. Please recall that, if σ is of the form $\forall\bar{\mathtt{X}}[U].\mathtt{X}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$, then $\sigma \preceq \mathtt{T}$ stands for $\exists\bar{\mathtt{X}}.(U \wedge \mathtt{X} = \mathtt{T})$. The process of constructing this constraint is informally referred to as "taking an instance of σ". It involves taking fresh copies of the type variables $\bar{\mathtt{X}}$, of the unification constraint U, and of the body $\mathtt{X}$. In the worst case, this process is just as inefficient as textually expanding the corresponding `let` construct in the program's source code, and leads to exponential time complexity (Mairson, Kanellakis, and Mitchell, 1991). In practice, however, the unification constraint U is often compact, because it was simplified before the environment frame $\mathsf{let}\ \mathtt{x} : \sigma\ \mathsf{in}\ []$ was created. which is why the solver usually performs well. (The creation of environment frames, performed by S-Pop-Let, is discussed below.) S-Solve-And discovers a conjunction. It arbitrarily chooses to explore the left branch first, and pushes a conjunction frame onto the stack, so as to record that the right branch should be explored afterwards. S-Solve-Ex discovers an existential quantifier and enters it, creating a new existential frame to record its existence. Similarly, S-Solve-Let discovers a let form and enters its left-hand side, creating a new let frame to record its existence. The choice of examining the left-hand side first is *not* arbitrary. Indeed, examining the right-hand side first would require creating an environment frame—but environment frames must contain *simplified* type schemes of the form $\forall\bar{\mathtt{X}}[U].\mathtt{X}$, whereas the type scheme $\forall\bar{\mathtt{X}}[D].\mathtt{T}$ is arbitrary. In other words, our strategy is to simplify type schemes prior to allowing them to be copied by S-Solve-Id, so as to avoid any duplication of effort. The side-conditions of S-Solve-Ex and S-Solve-Let may always be satisfied by suitable α-conversion of the left-hand state.

Rules S-Solve-Eq to S-Solve-Let may be referred to as *forward* rules, because they "move down into" the external constraint, causing the stack to grow. This process stops when the external constraint at hand becomes true. Then, part of the work has been finished, and the solver must examine the stack in order to determine what to do next. This task is performed by the last series of rules, which may be referred to as *backward* rules, because they "move back out", causing the stack to shrink, and possibly scheduling new external constraints for examination. These rules encode an analysis of the structure of the innermost stack frame. There are three cases, corresponding to conjunction, let, and environment frames. The case of existential stack frames need not be considered, because rules S-Ex-2 to S-Ex-4 allow either fusing them with let frames or floating them up to the outermost level, where they shall remain inert. S-Pop-And deals with conjunction frames. The frame is popped, and the external constraint that it carries is scheduled for examination. S-Pop-Env deals with environment frames. Because the right-hand side of the let construct at hand has been solved—that is, turned into a unification constraint U—it cannot contain an occurrence of $\mathtt{x}$. Furthermore, by assumption, $\exists\sigma$ is true. Thus, this environment frame is no longer useful: it is destroyed. The remaining rules deal with let frames. Roughly speaking, their purpose is to change the state $S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C]; U; \mathsf{true}$ into $S[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[U].\mathtt{T}\ \mathsf{in}\ []]; \mathsf{true}; C$, that is, to turn the current unification constraint U into a type scheme, turn the let frame into an environment frame, and schedule the right-hand side of the let construct (that is, the external constraint C) for examination. In fact, the process is more complex, because the type scheme $\forall\bar{\mathtt{X}}[U].\mathtt{T}$ must be *simplified* before becoming part of an environment frame. The simplification process is described by rules S-Name-2 to S-Pop-Let. In the following, we refer to type variables in $\bar{\mathtt{X}}$ as *young* and to type variables in $dtv(S) \setminus \bar{\mathtt{X}}$ as *old*. The former are the universal quantifiers of the type scheme that is being created; the latter are its free type variables.

S-Name-2 ensures that the body $\mathtt{T}$ of the type scheme that is being created is a type variable, as opposed to an arbitrary term. If it isn't, then it is replaced with a fresh variable $\mathtt{X}$, and the equation $\mathtt{X} = \mathtt{T}$ is added so as to recall that $\mathtt{X}$ stands for $\mathtt{T}$. Thus, the rule moves the term $\mathtt{T}$ into the current unification problem, where it potentially becomes subject to S-Name-1. This ensures that sharing is made explicit everywhere. S-Compress determines that the (young) type variable $\mathtt{Y}$ is an alias for the type variable $\mathtt{Z}$. Then, every free occurrence of $\mathtt{Y}$ other than its defining occurrence is replaced with $\mathtt{Z}$. In an actual implementation, this occurs transparently when the union-find algorithm performs *path compression* (Tarjan, 1975, 1979), provided we are careful never to create a link from a variable to a variable of higher rank. This requires making the unification algorithm aware of ranks, but is otherwise easily achieved. S-UnName determines that the (young) type variable $\mathtt{Y}$ has no occurrences other than its defining occurrence in the current type scheme. (This occurs, in particular, when S-Compress has just been applied.) Then, $\mathtt{Y}$ is suppressed altogether. In the particular case where the remaining multi-equation ϵ has cardinal 1, it may then be suppressed by S-Single. In other words, the combination of S-UnName and S-Single is able to suppress young unused type variables as well as the term that they stand for. This may, in turn, cause new type variables to become eligible for elimination by S-UnName. In fact, assuming the current unification constraint is acyclic, an inductive argument shows that every young type variable may be suppressed unless it is dominated either by $\mathtt{X}$ or by an old type variable. (In the setting of a regular tree model, it is possible to extend the rule so that young cycles that are not dominated either by $\mathtt{X}$ or by an old type variable are suppressed as well.) S-LetAll is a directed version of C-LetAll. It turns the young type variables $\bar{\mathtt{Y}}$ into old variables. How to tell whether $\exists\bar{\mathtt{X}}.U$ determines $\bar{\mathtt{Y}}$ is discussed later (see Lemma 1.8.7). Why S-LetAll is an interesting and important rule will be explained shortly. S-Pop-Let is meant to be applied when the current state has become a normal form with respect to S-Unify, S-Name-2, S-Compress, S-UnName, and S-LetAll, that is, when the type scheme that is about to be created is fully simplified. It splits the current unification constraint into two components U_1 and U_2, where U_1 is made up entirely of *old* variables—as expressed by the side-condition $\bar{\mathtt{X}} \# \mathit{ftv}(U_1)$—and U_2 constrains *young* variables only—as expressed by the side-condition $\exists\bar{\mathtt{X}}.U_2 \equiv \mathsf{true}$. Please note that U_2 may still contain free occurrences of old type variables, so the type scheme $\forall\bar{\mathtt{X}}[U_2].\mathtt{X}$ that appears on the right-hand side is not necessarily closed. It is not obvious why such a decomposition must exist; the proof of Lemma 1.8.11 sheds more light on this issue. Let us say, for now, that S-LetAll plays a role in guaranteeing its existence, whence part of its importance. Once the decomposition $U_1 \wedge U_2$ is obtained, the behavior of S-Pop-Let is simple. The unification constraint U_1 concerns old variables only, that is, variables that are not quantified in the current let frame; thus, it need not become part of the new type scheme, and may instead remain part of the current unification constraint. This is justified by C-LetAnd and C-InAnd* (see the proof of Lemma 1.8.10) and corresponds to the difference between hmx-Gen' and hmx-Gen discussed in Section 1.4. The unification constraint U_2, on the other hand, becomes part of the newly built type scheme $\forall\bar{\mathtt{X}}[U_2].\mathtt{X}$. The property $\exists\bar{\mathtt{X}}.U_2 \equiv \mathsf{true}$ guarantees that the newly created environment frame meets the requirements imposed on such frames. Please note that, the more type variables are considered old, the larger U_1 may become, and the smaller U_2. This is another reason why S-LetAll is interesting: by allowing more variables to be considered old, it decreases the size of the type scheme

$\forall\bar{\mathtt{X}}[U_2].\mathtt{X}$, making it cheaper to take instances of.

To complete our description of the constraint solver, there remains to explain how to decide when $\exists\bar{\mathtt{X}}.U$ determines $\bar{\mathtt{Y}}$, since this predicate occurs in the side-condition of S-LetAll. The following lemma describes two important situations where, by examining the structure of an equation, it is possible to discover that a constraint C *determines* some of its free type variables $\bar{\mathtt{Y}}$ (Definition 1.3.26). In the first situation, the type variables $\bar{\mathtt{Y}}$ are *equated* with or *dominated* by a distinct type variable $\mathtt{X}$ that occurs *free* in C. In that case, because the model is a free tree model, the values of the type variables $\bar{\mathtt{Y}}$ are determined by the value of $\mathtt{X}$—they are subtrees of it at specific positions. For instance, $\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2$ determines $\mathtt{Y}_1\mathtt{Y}_2$, while $\exists\mathtt{Y}_1.(\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2)$ determines $\mathtt{Y}_2$. In the second situation, the type variables $\bar{\mathtt{Y}}$ are equated with a term $\mathtt{T}$, *all* of whose free type variables are *free* in C. Again, the value of the type variables $\bar{\mathtt{Y}}$ is then determined by the values of the type variables $\mathit{ftv}(\mathtt{T})$—indeed, the term $\mathtt{T}$ itself defines a function that maps the latter to the former. For instance, $\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2$ determines $\mathtt{X}$, while $\exists\mathtt{Y}_1.(\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2)$ does not. In the second situation, no assumption is in fact made about the model. Please note that $\mathtt{X} = \mathtt{Y}_1 \rightarrow \mathtt{Y}_2$ determines $\mathtt{Y}_1\mathtt{Y}_2$ and determines $\mathtt{X}$, but does *not* simultaneously determine $\mathtt{X}\mathtt{Y}_1\mathtt{Y}_2$.

1.8.7 Lemma: Let $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$. Assume either ϵ is $\mathtt{X} = \epsilon'$, where $\mathtt{X} \notin \bar{\mathtt{X}}\bar{\mathtt{Y}}$ and $\bar{\mathtt{Y}} \subseteq \mathit{ftv}(\epsilon')$, or ϵ is $\bar{\mathtt{Y}} = \mathtt{T} = \epsilon'$, where $\mathit{ftv}(\mathtt{T}) \# \bar{\mathtt{X}}\bar{\mathtt{Y}}$. Then, $\exists\bar{\mathtt{X}}.(C \wedge \epsilon)$ determines $\bar{\mathtt{Y}}$. □

Proof: Let $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$ **(1)**. Let $\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.(C \wedge \epsilon)$ **(2)** and $\phi' \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \exists\bar{\mathtt{X}}.(C \wedge \epsilon)$ **(3)**, where ϕ and ϕ' coincide outside of $\bar{\mathtt{Y}}$. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)$ **(4)**. By (2), (4), CM-Exists, and CM-And, we obtain $\phi_1 \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \epsilon$ **(5)**, where ϕ and ϕ_1 coincide outside $\bar{\mathtt{X}}$. By CM-Predicate, (5) implies that all members of ϵ have the same image through ϕ_1. Similarly, exploiting (3) and (4), we find that all members of ϵ have the same image through ϕ'_1, where ϕ' and ϕ'_1 coincide outside $\bar{\mathtt{X}}$. Now, we claim that ϕ_1 and ϕ'_1 coincide on $\bar{\mathtt{Y}}$. Once the claim is established, by (1), there follows that ϕ and ϕ' must coincide on $\bar{\mathtt{Y}}$ as well, which is the goal. So, there only remains to establish the claim; we distinguish two subcases.

Subcase ϵ is $\mathtt{X} = \epsilon'$ and $\mathtt{X} \notin \bar{\mathtt{X}}\bar{\mathtt{Y}}$ **(6)** and $\bar{\mathtt{Y}} \subseteq \mathit{ftv}(\epsilon')$ **(7)**. Because ϕ_1 and ϕ'_1 coincide outside $\bar{\mathtt{X}}\bar{\mathtt{Y}}$ and by (6), we have $\phi_1(\mathtt{X}) = \phi'_1(\mathtt{X})$. As a result, all members of ϵ' have the same image through ϕ_1 and ϕ'_1. In a free tree model, where decomposition is valid, a simple inductive argument shows that ϕ_1 and ϕ'_1 must coincide on $\mathit{ftv}(\epsilon')$, hence—by (7)—also on $\bar{\mathtt{Y}}$.

Subcase ϵ is $\bar{\mathtt{Y}} = \mathtt{T} = \epsilon'$ and $\mathit{ftv}(\mathtt{T}) \# \bar{\mathtt{X}}\bar{\mathtt{Y}}$ **(8)**. Because ϕ_1 and ϕ'_1 coincide outside $\bar{\mathtt{X}}\bar{\mathtt{Y}}$ and by (8), we have $\phi_1(\mathtt{T}) = \phi'_1(\mathtt{T})$. Thus, for every $\mathtt{Y} \in \bar{\mathtt{Y}}$, we have $\phi_1(\mathtt{Y}) = \phi_1(\mathtt{T}) = \phi'_1(\mathtt{T}) = \phi'_1(\mathtt{Y})$. That is, ϕ_1 and ϕ'_1 coincide on $\bar{\mathtt{Y}}$. □

Thanks to Lemma 1.8.7, a straightforward implementation of S-LetAll comes to mind. The problem is, given a constraint $\exists\bar{\mathtt{X}}.U$, where U is a standard conjunction of multi-equations, to determine the greatest subset $\bar{\mathtt{Y}}$ of $\bar{\mathtt{X}}$ such that $\exists(\bar{\mathtt{X}} \setminus \bar{\mathtt{Y}}).U$ determines $\bar{\mathtt{Y}}$. By the first part of the lemma, it is safe for $\bar{\mathtt{Y}}$ to include all members of $\bar{\mathtt{X}}$ that are directly or indirectly dominated (with respect to U) by some free variable of $\exists\bar{\mathtt{X}}.U$. Those can be found, in time linear in the size of U, by a top-down traversal of the graph of $\prec_U$. By the second part of the lemma, it is safe to close $\bar{\mathtt{Y}}$ under the closure law $\mathtt{X} \in \bar{\mathtt{X}} \wedge (\forall\mathtt{Y} \quad \mathtt{Y} \prec_U \mathtt{X} \Rightarrow \mathtt{Y} \in \bar{\mathtt{Y}}) \Rightarrow \mathtt{X} \in \bar{\mathtt{Y}}$. That is, it is safe to also include all members of $\bar{\mathtt{X}}$ whose descendants (with respect to U) have already been found to be members of $\bar{\mathtt{Y}}$. This closure computation may be performed, again in linear time, by a bottom-up traversal of the graph of $\prec_U$. When U is acyclic, it is possible to show that this procedure is complete, that is, does compute the greatest subset $\bar{\mathtt{Y}}$ that meets our requirement. This is the topic of the following exercise.

1.8.8 Exercise [★★★, $\not\rightarrow$]: Assuming U is acyclic, prove that the above procedure computes the greatest subset $\bar{\mathtt{Y}}$ of $\bar{\mathtt{X}}$ such that $\exists(\bar{\mathtt{X}} \setminus \bar{\mathtt{Y}}).U$ determines $\bar{\mathtt{Y}}$. In the setting of a regular tree model, exhibit a satisfiable constraint U such that the above procedure is incomplete. Can you define a complete procedure in that setting? □

The above discussion has shown that when Y and Z are equated, if Y is young and Z is old, then S-LetAll allows making Y old as well. If binding information is encoded in terms of integer ranks, as suggested earlier, then this remark may be formulated as follows: when Y and Z are equated, if the rank of Y exceeds that of Z, then it may be decreased so that both ranks match. As a result, it is possible to attach ranks with *multi-equations*, rather than with variables. When two multi-equations are fused, the smaller rank is kept.

S-Solve-Let and S-Name-2 to S-Pop-Let are unnecessarily complex when x is assigned a *monotype* T, rather than an arbitrary type scheme $\forall\bar{\mathtt{X}}[D].\mathtt{T}$. In that case, the combined effect of these rules may be obtained directly via the following two new rules, which may be implemented in a more efficient way:

$$S; U; \mathsf{let}\ \mathtt{x} : \mathtt{T}\ \mathsf{in}\ C \quad\rightarrow\quad S[\exists\mathtt{X}.[]]; U \wedge \mathtt{X} = \mathtt{T}; \mathsf{let}\ \mathtt{x} : \mathtt{X}\ \mathsf{in}\ C \qquad \text{(S-Name-2-Mono)}$$
$$\text{if } \mathtt{X} \notin ftv(U, \mathtt{T}, C) \wedge \mathtt{T} \notin \mathcal{V}$$

$$S; U; \mathsf{let}\ \mathtt{x} : \mathtt{X}\ \mathsf{in}\ C \quad\rightarrow\quad S[\mathsf{let}\ \mathtt{x} : \mathtt{X}\ \mathsf{in}\ []]; U; C \qquad \text{(S-Solve-Let-Mono)}$$

If T isn't a variable, it is replaced with a fresh variable X, together with the equation X = T. This corresponds to the effect of S-Name-2. Then, we directly create an environment frame for $\mathtt{x}$, without bothering to create and discard a $\mathtt{let}$ frame, since there is no way the type scheme $\mathtt{X}$ may be further simplified.

Let us now state and establish the properties of the constraint solver. First, the reduction system is terminating, so it defines an algorithm.

1.8.9 LEMMA: The reduction system $\rightarrow$ is strongly normalizing. $\square$

Second, every rewriting step preserves the meaning of the constraint that the current state represents. We recall that the state $S; U; C$ is meant to represent the constraint $S[U \wedge C]$.

1.8.10 LEMMA: $S; U; C \rightarrow S'; U'; C'$ implies $S[U \wedge C] \equiv S'[U' \wedge C']$. $\square$

Proof: By examination of every rule.

∘ *Case* S-UNIFY. By Lemma 1.8.5.

∘ *Case* S-EX-1, S-EX-2, S-SOLVE-EX. By C-EXAND.

∘ *Case* S-EX-3. By C-LETEX.

∘ *Case* S-EX-4. By C-INEX.

∘ *Case* S-SOLVE-EQ, S-POP-AND. By C-DUP.

∘ *Case* S-SOLVE-ID. Because σ is of the form $\forall\bar{\mathtt{X}}[U].\mathtt{X}$, we have $fpi(\sigma) = \varnothing$. The result follows by C-INID.

∘ *Case* S-SOLVE-AND. By C-ANDAND.

∘ *Case* S-SOLVE-LET. By C-LETAND.

∘ *Case* S-NAME-2. By Definition 1.3.21 and C-NAMEEQ, $\mathtt{X} \notin ftv(U, \mathtt{T})$ implies $\mathsf{true} \Vdash \forall\bar{\mathtt{X}}[U].\mathtt{T} \equiv \forall\bar{\mathtt{X}}\mathtt{X}[U \wedge \mathtt{X} = \mathtt{T}].\mathtt{X}$. The result follows by Lemma 1.3.22.

∘ *Case* S-COMPRESS. Let $\theta = [\mathtt{Y} \mapsto \mathtt{Z}]$. By Definition 1.3.21 and C-NAMEEQ, $\mathtt{Y} \neq \mathtt{Z}$ implies $\mathsf{true} \Vdash \forall\bar{\mathtt{X}}\mathtt{Y}[\mathtt{Y} = \mathtt{Z} = \epsilon \wedge U].\mathtt{X} \equiv \forall\bar{\mathtt{X}}\mathtt{Y}[\mathtt{Y} \wedge \mathtt{Z} = \theta(\epsilon) \wedge \theta(U)].\theta(\mathtt{X})$. The result follows by Lemma 1.3.22.

∘ *Case* S-UNNAME. Using Lemma 1.3.18, it is straightforward to check that $\mathtt{Y} \notin ftv(\epsilon)$ implies $\exists\mathtt{Y}.(\mathtt{Y} = \epsilon) \equiv \epsilon$. The result follows by C-EXAND and C-LETEX.

∘ *Case* S-LETALL. By C-LETALL.

∘ *Case* S-POP-LET. By C-LETAND and C-INAND*.

∘ *Case* S-POP-ENV. By C-IN*, recalling that $\exists\sigma$ must be true. $\square$

Last, we classify the normal forms of the reduction system:

1.8.11 LEMMA: A normal form for the reduction system $\rightarrow$ is one of (i) $S; U; \mathtt{x} \preceq \mathtt{T}$, where $\mathtt{x} \notin dpi(S)$; (ii) $S; \mathsf{false}; \mathsf{true}$; or (iii) $\mathcal{X}; U; \mathsf{true}$, where $\mathcal{X}$ is an existential constraint context and U a satisfiable conjunction of multi-equations. $\square$

Proof: Because, by definition, $S; U; \mathsf{false}$ is not a valid state, a normal form for S-Solve-Eq, S-Solve-Id, S-Solve-And, S-Solve-Ex, and S-Solve-Let must be either an instance of the left-hand side of S-Solve-Id, with $\mathtt{x} \notin dpi(S)$, which corresponds to case (i), or of the form $S; U; \mathsf{true}$. Let us consider the latter case. Because $S; U; \mathsf{true}$ is a normal form with respect to S-Unify, by Lemma 1.8.6, U must be either false of the form $\mathcal{X}[U']$, where U' is a standard conjunction of multi-equations and, if the model is syntactic, U' is acyclic. The former case corresponds to (ii); thus, let us consider the latter case. Because $S; \mathcal{X}[U']; \mathsf{true}$ is a normal form with respect to S-Ex-1, the context $\mathcal{X}$ must in fact be empty, and U' is U. If S is an existential constraint context, then we are in situation (iii). Otherwise, because $S; U; \mathsf{true}$ is a normal form with respect to S-Ex-2, S-Ex-3, and S-Ex-4, the stack S does not end with an existential frame. Because $S; U; \mathsf{true}$ is a normal form with respect to S-Pop-And and S-Pop-Env, S must then be of the form $S'[\mathsf{let}\ \mathtt{x} : \forall\bar{\mathtt{X}}[[]].\mathtt{T}\ \mathsf{in}\ C]$. Because $S; U; \mathsf{true}$ is a normal form with respect to S-Name-2, $\mathtt{T}$ must be a type variable $\mathtt{X}$. Let us write U as $U_1 \wedge U_2$, where $\bar{\mathtt{X}} \# ftv(U_1)$, and where U_1 is maximal for this criterion. Then, consider a multi-equation $\epsilon \in U$. By the first part of Lemma 1.8.7, if one variable member of ϵ is free (that is, outside $\bar{\mathtt{X}}$), then $\exists\bar{\mathtt{X}}.U$ determines all other variables in $ftv(\epsilon)$. Because $S; U; \mathsf{true}$ is a normal form with respect to S-LetAll, all variables in $ftv(\epsilon)$ must then be free (that is, outside $\bar{\mathtt{X}}$). By definition of U_1, this implies $\epsilon \in U_1$. By contraposition, for every multi-equation $\epsilon \in U_2$, *all variable members of* ϵ *are in* $\bar{\mathtt{X}}$. Furthermore, let us recall that U_2 is a standard conjunction of multi-equations and, if the model is syntactic, U_2 is acyclic. We let the reader check that this implies $\exists\bar{\mathtt{X}}.U_2 \equiv \mathsf{true}$; the proof is a slight generalization of the last part of that of Lemma 1.8.6. Then, $S; U; \mathsf{true}$ is reducible via S-Pop-Let. This is a contradiction, so this last case cannot arise. □

In case (i), the constraint $S[U \wedge C]$ has a free program identifier $\mathtt{x}$, so it is not satisfiable. In other words, the source program contains an unbound program identifier. Such an error could of course be detected prior to constraint solving, if desired. In case (ii), the unification algorithm failed. By Lemma 1.3.30, the constraint $S[U \wedge C]$ is then false. In case (iii), the constraint $S[U \wedge C]$ is equivalent to $\mathcal{X}[U]$, where U is satisfiable, so it is satisfiable as well. Thus, each of the three classes of normal forms may be immediately identified as denoting success or failure. Thus, Lemmas 1.8.10 and 1.8.11 indeed prove that the algorithm is a constraint solver.

1.9 From ML-the-calculus to ML-the-programming-language

In this section, we explain how to extend the framework developed so far to accommodate operations on values of base type (such as integers), pairs, sums, references, and recursive function definitions. Then, we describe more complex extensions, namely algebraic data type definitions, pattern matching, and type annotations. Last, the issues associated with recursive types are briefly discussed. Exceptions are not discussed; the reader is referred to (TAPL Chapter 14).

Simple extensions

Many features of ML-the-programming-language may be introduced into ML-the-calculus by introducing new constants and extending $\xrightarrow{\delta}$ and Γ_0 appropriately. In each case, it is necessary to check that the requirements of Definition 1.7.6 are met, that is, the new initial environment faithfully reflects the nature of the new constants as well as the behavior of the new reduction rules. Below, we describe several such extensions in isolation.

1.9.1 Exercise [Integers, Recommended, ★★]: Integer literals and integer addition have been introduced and given an operational semantics in Examples 1.2.1, 1.2.2 and 1.2.4. Let us now introduce an isolated type constructor `int` of signature $\star$ and extend the initial environment Γ_0 with the bindings $\hat{n} : \mathtt{int}$, for every integer n, and $\hat{+} : \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int}$. Check that these definitions meet the requirements of Definition 1.7.6. □

1.9.2 Exercise [Booleans, Recommended, ★★, ↛]: Booleans and conditionals have been introduced and given an operational semantics in Exercise 1.2.6. Introduce an isolated type constructor `bool` to represent Boolean values and explain how to extend the initial environment. Check that your definitions meet the requirements of Definition 1.7.6. What is the constraint generation rule for the syntactic sugar $\mathtt{if}\ \mathtt{t}_0\ \mathtt{then}\ \mathtt{t}_1\ \mathtt{else}\ \mathtt{t}_2$? □

1.9.3 Exercise [Pairs, ★★, ↛]: Pairs and pair projections have been introduced and given an operational semantics in Examples 1.2.3 and 1.2.5. Let us now introduce an isolated type constructor $\times$ of signature $\star \otimes \star \Rightarrow \star$, covariant in both of its parameters, and extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rcl} (\cdot,\cdot) & : & \forall \mathtt{XY}.\mathtt{X} \rightarrow \mathtt{Y} \rightarrow \mathtt{X} \times \mathtt{Y} \\ \pi_1 & : & \forall \mathtt{XY}.\mathtt{X} \times \mathtt{Y} \rightarrow \mathtt{X} \\ \pi_2 & : & \forall \mathtt{XY}.\mathtt{X} \times \mathtt{Y} \rightarrow \mathtt{Y} \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.6. □

1.9.4 EXERCISE [SUMS, ★★, $\not\rightarrow$]: Sums have been introduced and given an operational semantics in Example 1.2.7. Let us now introduce an isolated type constructor $+$ of signature $\star \otimes \star \Rightarrow \star$, covariant in both of its parameters, and extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} \mathtt{inj}_1 : & \forall \mathtt{XY}.\mathtt{X} \rightarrow \mathtt{X} + \mathtt{Y} \\ \mathtt{inj}_2 : & \forall \mathtt{XY}.\mathtt{Y} \rightarrow \mathtt{X} + \mathtt{Y} \\ \mathtt{case} : & \forall \mathtt{XYZ}.(\mathtt{X} + \mathtt{Y}) \rightarrow (\mathtt{X} \rightarrow \mathtt{Z}) \rightarrow (\mathtt{Y} \rightarrow \mathtt{Z}) \rightarrow \mathtt{Z} \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.6. □

1.9.5 EXERCISE [REFERENCES, ★★★]: References have been introduced and given an operational semantics in Example 1.2.9. The type constructor ref has been introduced in Definition 1.7.4. Let us now extend the initial environment Γ_0 with the following bindings:

$$\begin{array}{rl} \mathtt{ref} : & \forall \mathtt{X}.\mathtt{X} \rightarrow \mathsf{ref}\,\mathtt{X} \\ ! : & \forall \mathtt{X}.\mathsf{ref}\,\mathtt{X} \rightarrow \mathtt{X} \\ := : & \forall \mathtt{X}.\mathsf{ref}\,\mathtt{X} \rightarrow \mathtt{X} \rightarrow \mathtt{X} \end{array}$$

Check that these definitions meet the requirements of Definition 1.7.6. □

1.9.6 EXERCISE [RECURSION, RECOMMENDED, ★★★]: The fixpoint combinator `fix` has been introduced and given an operational semantics in Example 1.2.10. Let us now extend the initial environment Γ_0 with the following binding:

$$\mathtt{fix} : \quad \forall \mathtt{XY}.((\mathtt{X} \rightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \rightarrow \mathtt{Y})) \rightarrow \mathtt{X} \rightarrow \mathtt{Y}$$

Check that these definitions meet the requirements of Definition 1.7.6. Recall how the `letrec` syntactic sugar was defined in Example 1.2.10, and check that this gives rise to the following constraint generation rule:

$$\begin{array}{rl} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{letrec}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{let}\ \mathtt{f} : \forall \mathtt{XY}[\mathsf{let}\ \mathtt{f} : \mathtt{X} \rightarrow \mathtt{Y}; \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{Y}]\!]].\mathtt{X} \rightarrow \mathtt{Y}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] \end{array}$$

Note the somewhat peculiar structure of this constraint: the program variable `f` is bound twice in it, with different type schemes. The constraint requires all occurrences of `f` within $\mathtt{t}_1$ to be assigned the *monomorphic* type $\mathtt{X} \rightarrow \mathtt{Y}$. This type is generalized and turned into a type scheme before inspecting $\mathtt{t}_2$, however, so every occurrence of `f` within $\mathtt{t}_2$ may receive a different type, as usual with `let`-polymorphism. A more powerful way of typechecking recursive function definitions is discussed in Section 1.10 (page 113). □

Algebraic data types

Exercises 1.9.3 and 1.9.4 have shown how to extend the language with binary, anonymous products and sums. These constructs are quite general, but still have several shortcomings. First, they are only binary, while we would like to have k-ary products and sums, for arbitrary $k \geq 0$. Such a generalization is of course straightforward. Second, more interestingly, their components must be referred to by numeric index (as in "please extract the *second* component of the pair"), rather than by name ("extract the component named `y`"). In practice, it is crucial to use names, because they make programs more readable and more robust in the face of changes. One could introduce a mechanism that allows defining names as syntactic sugar for numeric indices. That would help a little, but not much, because these names would not appear in *types*, which would still be made of anonymous products and sums. Third, in the absence of recursive types, products and sums do not have sufficient expressiveness to allow defining unbounded data structures, such as lists. Indeed, it is easy to see that every value whose type `T` is composed of base types (`int`, `bool`, *etc.*), products, and sums must have bounded size, where the bound $|\,\mathtt{T}\,|$ is a function of `T`. More precisely, up to a constant factor, we have $|\,\mathtt{int}\,| = |\,\mathtt{bool}\,| = 1$, $|\,\mathtt{T}_1 \times \mathtt{T}_2\,| = 1 + |\,\mathtt{T}_1\,| + |\,\mathtt{T}_2\,|$, and $|\,\mathtt{T}_1 + \mathtt{T}_2\,| = 1 + \max(|\,\mathtt{T}_1\,|, |\,\mathtt{T}_2\,|)$. The following example describes another facet of the same problem.

1.9.7 EXAMPLE: A list is either empty, or a pair of an element and another list. So, it seems natural to try and encode the type of lists as a sum of some arbitrary type (say, `unit`), on the one hand, and of a product of some element type and of the type of lists itself, on the other hand. With this encoding in mind, we can go ahead and write code—for instance, a function that computes the length of a list:

$$\mathtt{letrec\,length} = \lambda \mathtt{l}.\mathtt{case\;l}\;(\lambda\_.\hat{0})\;(\lambda \mathtt{z}.\hat{1} \mathbin{\hat{+}} \mathtt{length}\;(\pi_2\;\mathtt{z}))$$

We have used integers, pairs, sums, and the `letrec` construct introduced in the previous section. The code analyzes the list `l` using a `case` construct. If the left branch is taken, the list is empty, so 0 is returned. If the right branch is taken, then `z` becomes bound to a pair of some element and the tail of the list. The latter is obtained using the projection operator π_2. Its length is computed using a recursive call to `length` and incremented by 1. This code makes perfect sense. However, applying the constraint generation and constraint solving algorithms eventually leads to an equation of the form $\mathtt{X} = \mathtt{Y} + (\mathtt{Z} \times \mathtt{X})$, where `X` stands for the type of `l`. This equation accurately reflects our encoding of the type of lists. However, in a syntactic model, it has no solution, so our definition of `length` is ill-typed. It is possible to adopt a free regular tree model,thus introducing *equirecursive* types into the system (TAPL Chapter 20); however, there are good reasons not to do so (page 106). □

To work around this problem, ML-the-programming-language offers *algebraic data type* definitions, whose elegance lies in the fact that, while representing only a modest theoretical extension, they do solve the three problems mentioned above. An algebraic data type may be viewed as an *abstract type* that is declared to be *isomorphic* to a (k-ary) product or sum type with named components. The type of each component is declared as well, and may refer to the algebraic data type that is being defined: thus, algebraic data types are *isorecursive* (TAPL Chapter 20). In order to allow sufficient flexibility when declaring the type of each component, algebraic data type definitions may be *parameterized* by a number of type variables. Last, in order to allow the description of complex data structures, it is necessary to allow several algebraic data types to be defined at once; the definitions may then be *mutually recursive*. In fact, in order to simplify this formal presentation, we assume that *all* algebraic data types are defined at once at the beginning of the program. This decision is of course at odds with modular programming, but will not otherwise be a problem.

In the following, $\mathtt{D}$ ranges over a set of *data types*. We assume that data types form a subset of type constructors. We require each of them to be isolated and to have a signature of the form $\vec{\kappa} \Rightarrow \star$. Furthermore, ℓ ranges over a set $\mathcal{L}$ of *labels*, which we use indifferently as *data constructors* and as *record labels*. An *algebraic data type definition* is either a *variant type* definition or a *record type* definition, whose respective forms are

$$\mathtt{D}\,\vec{\mathtt{X}} \approx \sum_{i=1}^{k} \ell_i : \mathtt{T}_i \qquad \text{and} \qquad \mathtt{D}\,\vec{\mathtt{X}} \approx \prod_{i=1}^{k} \ell_i : \mathtt{T}_i.$$

In either case, k must be nonnegative. If $\mathtt{D}$ has signature $\vec{\kappa} \Rightarrow \star$, then the type variables $\vec{\mathtt{X}}$ must have kind $\vec{\kappa}$. Every $\mathtt{T}_i$ must have kind $\star$. We refer to $\bar{\mathtt{X}}$ as the *parameters* and to $\vec{\mathtt{T}}$ (the vector formed by $\mathtt{T}_1, \ldots, \mathtt{T}_k$) as the *components* of the definition. The parameters are bound within the components, and the definition must be closed, that is, $ftv(\vec{\mathtt{T}}) \subseteq \bar{\mathtt{X}}$ must hold. Last, for an algebraic data type definition to be valid, the behavior of the type constructor $\mathtt{D}$ with respect to subtyping must match its definition. This requirement is clarified below.

1.9.8 Definition: Consider an algebraic data type definition whose parameters and components are respectively $\vec{\mathtt{X}}$ and $\vec{\mathtt{T}}$. Let $\vec{\mathtt{X}}'$ and $\vec{\mathtt{T}}'$ be their images under an arbitrary renaming. Then, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \Vdash \vec{\mathtt{T}} \leq \vec{\mathtt{T}}'$ must hold. □

The above requirement bears on the definition of subtyping in the model. The idea is, since $\mathtt{D}\,\vec{\mathtt{X}}$ is declared to be isomorphic to (a sum or a product of)

$\vec{\mathtt{T}}$, whenever two types built with $\mathtt{D}$ are comparable, their unfoldings should be comparable as well. The reverse entailment assertion is not required for type soundness, and it is sometimes useful to declare algebraic data types that do not validate it—so-called *phantom types* (Fluet and Pucella, 2002). Note that the requirement may always be satisfied by making the type constructor $\mathtt{D}$ *invariant* in all of its parameters. Indeed, in that case, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}'$ entails $\vec{\mathtt{X}} = \vec{\mathtt{X}}'$, which must entail $\vec{\mathtt{T}} = \vec{\mathtt{T}}'$ since $\vec{\mathtt{T}}'$ is precisely $[\vec{\mathtt{X}} \mapsto \vec{\mathtt{X}}']\vec{\mathtt{T}}$. In an equality free tree model, every type constructor is naturally invariant, so the requirement is trivially satisfied. In other settings, however, it is often possible to satisfy the requirement of Definition 1.9.8 while assigning $\mathtt{D}$ a less restrictive variance. The following example illustrates such a case.

1.9.9 Example: Let list be a data type of signature $\star \Rightarrow \star$. Let $\mathtt{Nil}$ and $\mathtt{Cons}$ be data constructors. Then, the following is a definition of list as a variant type:

$$\mathsf{list}\,\mathtt{X} \approx \Sigma\,(\mathtt{Nil} : \mathtt{unit}; \mathtt{Cons} : \mathtt{X} \times \mathsf{list}\,\mathtt{X})$$

Because data types form a subset of type constructors, it is valid to form the type $\mathsf{list}\,\mathtt{X}$ in the right-hand side of the definition, even though we are still in the process of defining the meaning of list. In other words, data type definitions may be recursive. However, because $\approx$ is not interpreted as equality, the type $\mathsf{list}\,\mathtt{X}$ is *not* a recursive type: it is nothing but an application of the unary type constructor list to the type variable $\mathtt{X}$. To check that the definition of list satisfies the requirement of Definition 1.9.8, we must ensure that

$$\mathsf{list}\,\mathtt{X} \leq \mathsf{list}\,\mathtt{X}' \Vdash \mathtt{unit} \leq \mathtt{unit} \wedge \mathtt{X} \times \mathsf{list}\,\mathtt{X} \leq \mathtt{X}' \times \mathsf{list}\,\mathtt{X}'$$

holds. This assertion is equivalent to $\mathsf{list}\,\mathtt{X} \leq \mathsf{list}\,\mathtt{X}' \Vdash \mathtt{X} \leq \mathtt{X}'$. To satisfy the requirement, it is sufficient to make list a *covariant* type constructor, that is, to define subtyping in the model so that $\mathsf{list}\,\mathtt{X} \leq \mathsf{list}\,\mathtt{X}' \equiv \mathtt{X} \leq \mathtt{X}'$ holds.

Let tree be a data type of signature $\star \Rightarrow \star$. Let $\mathtt{root}$ and $\mathtt{sons}$ be record labels. Then, the following is a definition of tree as a record type:

$$\mathsf{tree}\,\mathtt{X} \approx \Pi\,(\mathtt{root} : \mathtt{X}; \mathtt{sons} : \mathsf{list}\,(\mathsf{tree}\,\mathtt{X}))$$

This definition is again recursive, and relies on the previous definition. Because list is covariant, it is straightforward to check that the definition of tree is valid if tree is made a covariant type constructor as well. □

1.9.10 Exercise [★★, $\not\rightarrow$]: Consider a nonrecursive algebraic data type definition, where the variance of every type constructor that appears on the right-hand side is known. Can you systematically determine, for each of the parameters, the least restrictive variance that makes the definition valid? Generalize this procedure to the case of recursive and mutually recursive algebraic data type definitions. □

A *prologue* is a set of algebraic data type definitions, where each data type is defined at most once and where each data constructor or record label appears at most once. A *program* is a pair of a prologue and an expression. The effect of a prologue is to enrich the programming language with new constants. That is, a variant type definition extends the operational semantics with several injections and a `case` construct, as in Example 1.2.7. A record type definition extends it with a record formation construct and several projections, as in Examples 1.2.3 and 1.2.5. In either case, the initial typing environment Γ_0 is extended with information about these new constants. Thus, algebraic data type definitions might be viewed as a simple configuration language that allows specifying in which instance of ML-the-calculus the expression that follows the prologue should be typechecked and interpreted. Let us now give a precise account of this phenomenon.

To begin, suppose the prologue contains the definition $\mathtt{D}\,\vec{\mathtt{X}} \approx \sum_{i=1}^{k} \ell_i : \mathtt{T}_i$. Then, for each $i \in \{1, \ldots, k\}$, a constructor of arity 1, named ℓ_i, is introduced. Furthermore, a destructor of arity $k+1$, named $\mathtt{case}_\mathtt{D}$, is introduced. When $k > 0$, it is common to write $\mathtt{case}\,\mathtt{t}\,[\ell_i : \mathtt{t}_i]_{i=1}^{k}$ for the application $\mathtt{case}_\mathtt{D}\,\mathtt{t}\,\mathtt{t}_1\,\ldots\,\mathtt{t}_n$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$\mathtt{case}\,(\ell_i\,\mathtt{v})\,[\ell_j : \mathtt{v}_j]_{j=1}^{k} \xrightarrow{\delta} \mathtt{v}_i\,\mathtt{v} \qquad \text{(R-Alg-Case)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall\bar{\mathtt{X}}.\mathtt{T}_i \to \mathtt{D}\,\vec{\mathtt{X}}$. It is further extended with the binding $\mathtt{case}_\mathtt{D} : \forall\bar{\mathtt{X}}\mathtt{Z}.\mathtt{D}\,\vec{\mathtt{X}} \to (\mathtt{T}_1 \to \mathtt{Z}) \to \ldots (\mathtt{T}_k \to \mathtt{Z}) \to \mathtt{Z}$.

Now, suppose the prologue contains the definition $\mathtt{D}\,\vec{\mathtt{X}} \approx \prod_{i=1}^{k} \ell_i : \mathtt{T}_i$. Then, for each $i \in \{1, \ldots, k\}$, a destructor of arity 1, named ℓ_i, is introduced. Furthermore, a constructor of arity k, named $\mathtt{make}_\mathtt{D}$, is introduced. It is common to write $\mathtt{t}.\ell$ for the application $\ell\,\mathtt{t}$ and, when $k > 0$, to write $\{\ell_i = \mathtt{t}_i\}_{i=1}^{k}$ for the application $\mathtt{make}_\mathtt{D}\,\mathtt{t}_1\,\ldots\,\mathtt{t}_k$. The operational semantics is extended with the following reduction rules, for $i \in \{1, \ldots, k\}$:

$$(\{\ell_j = \mathtt{v}_j\}_{j=1}^{k}).\ell_i \xrightarrow{\delta} \mathtt{v}_i \qquad \text{(R-Alg-Proj)}$$

For each $i \in \{1, \ldots, k\}$, the initial environment is extended with the binding $\ell_i : \forall\bar{\mathtt{X}}.\mathtt{D}\,\vec{\mathtt{X}} \to \mathtt{T}_i$. It is further extended with the binding $\mathtt{make}_\mathtt{D} : \forall\bar{\mathtt{X}}.\mathtt{T}_1 \to \ldots \to \mathtt{T}_k \to \mathtt{D}\,\vec{\mathtt{X}}$.

1.9.11 Example: The effect of defining list (Example 1.9.9) is to make `Nil` and `Cons` data constructors of arity 1 and to introduce a binary destructor $\mathtt{case}_{\mathsf{list}}$. The definition also extends the initial environment as follows:

$$\begin{aligned}
\mathtt{Nil} :&\quad \forall\mathtt{X}.\mathtt{unit} \to \mathsf{list}\,\mathtt{X} \\
\mathtt{Cons} :&\quad \forall\mathtt{X}.\mathtt{X} \times \mathsf{list}\,\mathtt{X} \to \mathsf{list}\,\mathtt{X} \\
\mathtt{case}_{\mathsf{list}} :&\quad \forall\mathtt{X}\mathtt{Z}.\mathsf{list}\,\mathtt{X} \to (\mathtt{unit} \to \mathtt{Z}) \to (\mathtt{X} \times \mathsf{list}\,\mathtt{X} \to \mathtt{Z}) \to \mathtt{Z}
\end{aligned}$$

Thus, the value $\texttt{Cons}(\hat{0}, \texttt{Nil}())$, an integer list of length 1, has type $\mathsf{list}\,\texttt{int}$. A function that computes the length of a list may now be written as follows:

$$\texttt{letrec}\,\texttt{length} = \lambda\texttt{l}.\mathsf{case}\,\texttt{l}\,[\,\texttt{Nil} : \lambda\_.\hat{0} \mid \texttt{Cons} : \lambda\texttt{z}.\hat{1} \mathbin{\hat{+}} \texttt{length}\,(\pi_2\,\texttt{z})\,]$$

Recall that this notation is syntactic sugar for

$$\texttt{letrec}\,\texttt{length} = \lambda\texttt{l}.\mathsf{case}_{\mathsf{list}}\,\texttt{l}\,(\lambda\_.\hat{0})\,(\lambda\texttt{z}.\hat{1} \mathbin{\hat{+}} \texttt{length}\,(\pi_2\,\texttt{z}))$$

The difference with the code in Example 1.9.7 appears minimal: the case construct is now annotated with the data type list. As a result, the type inference algorithm employs the type scheme assigned to $\texttt{case}_{\mathsf{list}}$, which is derived from the definition of list, instead of the type scheme assigned to the anonymous case construct, given in Exercise 1.9.4. This is good for a couple of reasons. First, the former is more informative than the latter, because it contains the type $\texttt{T}_i$ associated with the data constructor ℓ_i. Here, for instance, the generated constraint requires the type of z to be $\texttt{X} \times \mathsf{list}\,\texttt{X}$ for some X, so a good error message would be given if a mistake was made in the second branch, such as omitting the use of π_2. Second, and more fundamentally, *the code is now well-typed*, even in the absence of recursive types. In Example 1.9.7, a cyclic equation was produced because case required the type of l to be a sum type and because a sum type carries the types of its left and right branches as subterms. Here, instead, $\texttt{case}_{\mathsf{list}}$ requires l to have type $\mathsf{list}\,\texttt{X}$ for some X. This is an abstract type: it does not explicitly contain the types of the branches. As a result, the generated constraint no longer involves a cyclic equation. It is, in fact, satisfiable; the reader may check that length has type $\forall\texttt{X}.\mathsf{list}\,\texttt{X} \rightarrow \texttt{int}$, as expected. □

Example 1.9.11 stresses the importance of using *declared, abstract* types, as opposed to *anonymous, concrete* sum or product types, in order to obviate the need for recursive types. The essence of the trick lies in the fact that the type schemes associated with operations on algebraic data types implicitly *fold* and *unfold* the data type's definition. More precisely, let us recall the type scheme assigned to the i^{th} injection in the setting of (k-ary) anonymous sums: it is $\forall\texttt{X}_1 \ldots \texttt{X}_k.\texttt{X}_i \rightarrow \texttt{X}_1 + \ldots + \texttt{X}_k$, or, more concisely, $\forall\texttt{X}_1 \ldots \texttt{X}_k.\texttt{X}_i \rightarrow \sum_{i=1}^{k} \texttt{X}_i$. By instantiating each $\texttt{X}_i$ with $\texttt{T}_i$ and generalizing again, we find that a more specific type scheme is $\forall\bar{\texttt{X}}.\texttt{T}_i \rightarrow \sum_{i=1}^{k} \texttt{T}_i$. Perhaps this could have been the type scheme assigned to ℓ_i? Instead, however, it is $\forall\bar{\texttt{X}}.\texttt{T}_i \rightarrow \texttt{D}\,\vec{\texttt{X}}$. We now realize that this type scheme not only reflects the operational behavior of the i^{th} injection, but also *folds* the definition of the algebraic data type D by turning the anonymous sum $\sum_{i=1}^{k} \texttt{T}_i$—which forms the definition's right-hand side—into the parameterized abstract type $\texttt{D}\,\vec{\texttt{X}}$—which is the definition's left-hand side. Conversely, the type scheme assigned to $\texttt{case}_{\texttt{D}}$ *unfolds* the definition. The situation is identical in the case of record types: in either case, *constructors fold, destructors unfold.* In other words, occurrences of data constructors and record labels in the code may be viewed as explicit instructions for the typechecker to fold or unfold an algebraic data type definition. This mechanism is characteristic of *isorecursive* types.

1.9.12 Exercise [★, $\nrightarrow$]: For a fixed k, check that all of the machinery associated with k-ary anonymous products—that is, constructors, destructors, reduction rules, and extensions to the initial typing environment—may be viewed as the result of a single algebraic data type definition. Conduct a similar check in the case of k-ary anonymous sums. □

1.9.13 Exercise [★★★, $\nrightarrow$]: Check that the above definitions meet the requirements of Definition 1.7.6. □

1.9.14 Exercise [★★★, $\nrightarrow$]: For sake of simplicity, we have assumed that data constructors are always of arity one. It is indeed possible to allow data constructors of any arity and define variants as $\mathtt{D}\,\vec{\mathtt{X}} \approx \sum_{i=1}^{k} \ell_i : \vec{\mathtt{T}}_i$. For instance, the definition of list could then be $\mathsf{list}\,\mathtt{X} \approx \Sigma\,(\mathtt{Nil}; \mathtt{Cons} : \mathtt{X} \times \mathsf{list}\,\mathtt{X})$ and for instance $\mathtt{Cons}(\hat{0}, \mathtt{Nil})$ would be a list value. Make the necessary changes in the definitions above and check that they still meet the requirements of Definition 1.7.6. □

In this formal presentation of algebraic data types, we have assumed that all algebraic data type definitions are known before the program is typechecked. This simplifying assumption is forced on us by the fact that we interpret constraints in a fixed model, that is, we assume a fixed universe of types. In practice, programming languages have *module systems*, which allow distinct modules to have distinct, partial views of the universe of types. Then, it becomes possible for each module to come with its own data type definitions. Interestingly, it is even possible, in principle, to split the definition of a single data type over several modules, yielding *extensible* algebraic data types. For instance, module A might declare the existence of a parameterized variant type $\mathtt{D}\,\vec{\mathtt{X}}$, without giving its components. Later on, module B might define a component $\ell : \mathtt{T}$, where $\mathit{ftv}(\mathtt{T}) \subseteq \bar{\mathtt{X}}$. Such a definition makes ℓ a unary constructor with type scheme $\forall\bar{\mathtt{X}}.\mathtt{T} \rightarrow \mathtt{D}\,\vec{\mathtt{X}}$, as before. It becomes impossible, however, to introduce a destructor $\mathtt{case}_{\mathtt{D}}$, because the definition of an extensible variant type can never be assumed to be complete—other, unknown modules might extend it further. To compensate for its absence, one may supplement every constructor ℓ with a destructor ℓ^{-1}, whose semantics is given by $\ell^{-1}\ (\ell\ \mathtt{v})\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_1\ \mathtt{v}$ and $\ell^{-1}\ (\ell'\ \mathtt{v})\ \mathtt{v}_1\ \mathtt{v}_2 \xrightarrow{\delta} \mathtt{v}_2\ (\ell'\ \mathtt{v})$ when $\ell \neq \ell'$, and whose type scheme is $\forall\bar{\mathtt{X}}\mathtt{Z}.\mathtt{D}\,\vec{\mathtt{X}} \rightarrow (\mathtt{T} \rightarrow \mathtt{Z}) \rightarrow (\mathtt{D}\,\vec{\mathtt{X}} \rightarrow \mathtt{Z}) \rightarrow \mathtt{Z}$. When pattern matching is available, ℓ^{-1} may in fact be defined in the language. ML-the-programming-language does not offer extensible algebraic data types as a language feature, but does have one built-in extensible variant type, namely the type `exn` of exceptions. Thus, it is possible to define new constructors for the type `exn` within any module. The price of this extra flexibility is that no exhaustive case analysis on values of type `exn` is possible.

One significant drawback of algebraic data type definitions resides in the fact that a label ℓ cannot be *shared* by two distinct variant or record type definitions. Indeed, every algebraic data type definition extends the calculus with new constants. Strictly speaking, our presentation does not allow a single constant `c` to be associated with two distinct definitions. Even if we did allow such a collision, the initial environment would contain two bindings for `c`, one of which would then become inaccessible. This phenomenon arises in actual implementations of ML-the-programming-language, where a new algebraic data type definition may hide some of the data constructors or record labels introduced by a previous definition. An elegant solution to this lack of expressiveness is discussed in Section 1.11.

Pattern matching

Our presentation of products, sums and algebraic data types has remained within the setting of ML-the-calculus: that is, data structures have been built out of constructors, while the case analysis and record access operations have been viewed as destructors. Some syntactic sugar has been used to recover standard notations. The language is now expressive enough to allow defining and manipulating complex data structures, such as lists and trees. Yet, experience shows that programming in such a language is still somewhat cumbersome. Indeed, case analysis and record access are low-level operations: the former allows inspecting a tag and branching, while the latter allows dereferencing a pointer. In practice, one often needs to carry out more complex tasks, such as determining whether a data structure has a certain shape or whether two data structures have comparable shapes. Currently, the only way to carry out these tasks is to program an explicit sequence of low-level operations. It would be much preferable to extend the language so that it becomes directly possible to describe shapes, called *patterns*, and so that checking whether a patterns *matches* a value becomes an elementary operation. ML-the-programming-language offers this feature, called *pattern matching*. Although pattern matching may be added to ML-the-calculus by introducing a family of destructors, we rather choose to extend the calculus with a new `match` construct, which subsumes the existing `let` construct. This approach appears somewhat simpler and more powerful. We now carry out this

$\mathtt{p} ::=$		*Patterns:*
	$\_$	*Wildcard*
	$\mathtt{z}$	*Variable*
	$\mathtt{c}\,\mathtt{p}_1 \ \dots\ \mathtt{p}_k$	*Data*
		$\mathtt{c} \in \mathcal{Q}^+ \wedge k = a(\mathtt{c})$
	$\mathtt{p} \wedge \mathtt{p}$	*Conjunction*
	$\mathtt{p} \vee \mathtt{p}$	*Disjunction*

Pattern matching

$$
\begin{aligned}
[\_ \mapsto \mathtt{v}] &= \varnothing \\
[\mathtt{c}\,\mathtt{p}_1 \ \dots\ \mathtt{p}_k \mapsto \mathtt{c}\,\mathtt{v}_1 \ \dots\ \mathtt{v}_k] & \\
&= [\mathtt{p}_1 \mapsto \mathtt{v}_1] \otimes \dots \otimes [\mathtt{p}_k \mapsto \mathtt{v}_k] \\
[\mathtt{p}_1 \wedge \mathtt{p}_2 \mapsto \mathtt{v}] &= [\mathtt{p}_1 \mapsto \mathtt{v}] \otimes [\mathtt{p}_2 \mapsto \mathtt{v}] \\
[\mathtt{p}_1 \vee \mathtt{p}_2 \mapsto \mathtt{v}] &= [\mathtt{p}_1 \mapsto \mathtt{v}] \oplus [\mathtt{p}_2 \mapsto \mathtt{v}]
\end{aligned}
$$

Figure 1-13: Patterns and pattern matching

extension.

Let us first define the syntax of patterns (Figure 1-13) and describe (informally, for now) which values they match. To a pattern $\mathtt{p}$, we associate a set of *defined program variables* $dpi(\mathtt{p})$, whose definition appears in the text that follows. The pattern $\mathtt{p}$ is well-formed if and only if $dpi(\mathtt{p})$ is defined. To begin, the wildcard $\_$ is a pattern, which matches every value and binds no variables. We let $dpi(\_) = \varnothing$. Although the wildcard may be viewed as an anonymous variable, and we have done so thus far, it is now simpler to view it as a distinct pattern. A program variable $\mathtt{z}$ is also a pattern, which matches every value and binds $\mathtt{z}$ to the matched value. We let $dpi(\mathtt{z}) = \{\mathtt{z}\}$. Next, if $\mathtt{c}$ is a constructor of arity k, then $\mathtt{c}\,\mathtt{p}_1 \ \dots\ \mathtt{p}_k$ is a pattern, which matches $\mathtt{c}\,\mathtt{v}_1 \ \dots\ \mathtt{v}_k$ when $\mathtt{p}_i$ matches $\mathtt{v}_i$ for every $i \in \{1, \dots, k\}$. We let $dpi(\mathtt{c}\,\mathtt{p}_1 \ \dots\ \mathtt{p}_k) = dpi(\mathtt{p}_1) \uplus \dots \uplus dpi(\mathtt{p}_k)$. That is, the pattern $\mathtt{c}\,\mathtt{p}_1 \ \dots\ \mathtt{p}_k$ is well-formed when $\mathtt{p}_1, \dots, \mathtt{p}_k$ define *disjoint* sets of variables. This condition rules out *nonlinear* patterns such as $(\mathtt{z}, \mathtt{z})$. Defining the semantics of such a pattern would require a notion of equality at every type, which introduces various complications, so it is commonly considered ill-formed. The pattern $\mathtt{p}_1 \wedge \mathtt{p}_2$ matches all values that both $\mathtt{p}_1$ and $\mathtt{p}_2$ match. It is commonly used with $\mathtt{p}_2$ a program variable: then, it allows examining the shape of a value and binding a name to it at the same time. Again, we define $dpi(\mathtt{p}_1 \wedge \mathtt{p}_2) = dpi(\mathtt{p}_1) \uplus dpi(\mathtt{p}_2)$. The pattern $\mathtt{p}_1 \vee \mathtt{p}_2$ matches all values that either $\mathtt{p}_1$ or $\mathtt{p}_2$ matches. We define $dpi(\mathtt{p}_1 \vee \mathtt{p}_2) = dpi(\mathtt{p}_1) = dpi(\mathtt{p}_2)$. That is, the pattern $\mathtt{p}_1 \vee \mathtt{p}_2$ is well-formed when $\mathtt{p}_1$ and $\mathtt{p}_2$ define the *same* variables. Thus, $(\mathtt{inj}_1\ \mathtt{z}) \vee (\mathtt{inj}_2\ \mathtt{z})$ is a well-formed pattern, which binds $\mathtt{z}$ to the component of a binary sum, without regard for its tag. However, $(\mathtt{inj}_1\ \mathtt{z}_1) \vee (\mathtt{inj}_2\ \mathtt{z}_2)$ is ill-formed, because one cannot statically predict whether it defines $\mathtt{z}_1$ or $\mathtt{z}_2$.

Let us now formally define whether a pattern $\mathtt{p}$ matches a value $\mathtt{v}$ and how the variables in $dpi(\mathtt{p})$ become bound to values in the process. This is done by introducing a *generalized substitution*, written $[\mathtt{p} \mapsto \mathtt{v}]$, which is either

$\mathtt{t}$ ::=	... *Expressions:*	*Reduction rules*	
	$\texttt{match t with } (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^k$	$\texttt{match v with } (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^k \longrightarrow \bigoplus_{i=1}^k [\mathtt{p}_i \mapsto \mathtt{v}]\mathtt{t}_i$	(R-Match)
$\mathcal{E}$ ::=	... *Evaluation Contexts:*		
	$\texttt{match } \mathcal{E} \texttt{ with } (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^k$		

Figure 1-14: Extended syntax and semantics of ML-the-calculus

undefined or a substitution of values for the program variables in $dpi(\mathtt{p})$. If the former, then $\mathtt{p}$ does not match $\mathtt{v}$. If the latter, then $\mathtt{p}$ matches $\mathtt{v}$ and, for every $\mathtt{z} \in dpi(\mathtt{p})$, the variable $\mathtt{z}$ becomes bound to the value $[\mathtt{p} \mapsto \mathtt{v}]\mathtt{z}$. Of course, when $\mathtt{p}$ is a variable $\mathtt{z}$, the generalized substitution $[\mathtt{z} \mapsto \mathtt{v}]$ is defined and coincides with the substitution $[\mathtt{z} \mapsto \mathtt{v}]$, which justifies our abuse of notation. To construct generalized substitutions, we use two simple combinators. First, when $dpi(\mathtt{p}_1)$ and $dpi(\mathtt{p}_2)$ are disjoint, $[\mathtt{p}_1 \mapsto \mathtt{v}_1] \otimes [\mathtt{p}_2 \mapsto \mathtt{v}_2]$ stands for the set-theoretic union of $[\mathtt{p}_1 \mapsto \mathtt{v}_1]$ and $[\mathtt{p}_2 \mapsto \mathtt{v}_2]$, if both are defined, and is undefined otherwise. We use this combinator to ensure that $\mathtt{p}_1$ matches $\mathtt{v}_1$ and $\mathtt{p}_2$ matches $\mathtt{v}_2$ and to combine the two corresponding sets of bindings. Second, when o_1 and o_2 are two possibly undefined mathematical objects that belong to the same space when defined, $o_1 \oplus o_2$ stands for o_1, if it is defined, and for o_2 otherwise—that is, $\oplus$ is an angelic choice operator with a left bias. In particular, when $dpi(\mathtt{p}_1)$ and $dpi(\mathtt{p}_2)$ coincide, $[\mathtt{p}_1 \mapsto \mathtt{v}_1] \oplus [\mathtt{p}_2 \mapsto \mathtt{v}_2]$ stands for $[\mathtt{p}_1 \mapsto \mathtt{v}_1]$, if it is defined, and for $[\mathtt{p}_2 \mapsto \mathtt{v}_2]$ otherwise. We use this combinator to ensure that $\mathtt{p}_1$ matches $\mathtt{v}_1$ or $\mathtt{p}_2$ matches $\mathtt{v}_2$ and to retain the corresponding set of bindings. The full definition of generalized substitutions, which relies on these combinators, appears in Figure 1-13. It reflects the informal presentation of the previous paragraph.

Once patterns and pattern matching are defined, it is straightforward to extend the syntax and operational semantics of ML-the-calculus. We enrich the syntax of expressions with a new construct, $\texttt{match t with } (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^k$, where $k \geq 1$. It consists of a term $\mathtt{t}$ and a nonempty, ordered list of clauses, each of which is composed of a pattern $\mathtt{p}_i$ and a term $\mathtt{t}_i$. The syntax of evaluation contexts is extended as well, so that the term $\mathtt{t}$ that is being examined is first reduced to a value $\mathtt{v}$. The operational semantics is extended with a new rule, R-Match, which states that $\texttt{match v with } (\mathtt{p}_i\ .\ \mathtt{t}_i)_{i=1}^k$ reduces to $[\mathtt{p}_i \mapsto \mathtt{v}]\mathtt{t}_i$, where i is the least element of $\{1, \ldots, k\}$ such that $\mathtt{p}_i$ matches $\mathtt{v}_i$. Technically, $\bigoplus_{i=1}^k [\mathtt{p}_i \mapsto \mathtt{v}]\mathtt{t}_i$ stands for $[\mathtt{p}_1 \mapsto \mathtt{v}]\mathtt{t}_1 \oplus \ldots \oplus [\mathtt{p}_k \mapsto \mathtt{v}]\mathtt{t}_k$, so that the reduct is the first term that is defined in this sequence.

As far as semantics is concerned, the $\mathtt{match}$ construct may be viewed as a generalization of the `let` construct. Indeed, $\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ may now be viewed as syntactic sugar for $\mathtt{match}\,\mathtt{t}_1\ \mathtt{with}\,\mathtt{z}\,.\,\mathtt{t}_2$, that is, a `match` construct with a single clause and a variable pattern. Then, R-Let becomes a special case of R-Match.

It is pleasant to introduce some more syntactic sugar. We write $\lambda(\mathtt{p}_i.\mathtt{t}_i)_{i=1}^k$ for $\lambda\mathtt{z}.\mathtt{match}\ \mathtt{z}\ \mathtt{with}\ (\mathtt{p}_i\,.\,\mathtt{t}_i)_{i=1}^k$, where `z` is fresh for $(\mathtt{p}_i.\mathtt{t}_i)_{i=1}^k$. Thus, it becomes possible to define functions by cases—a common idiom in ML-the-programming-language.

1.9.15 Example: Using pattern matching, a function that computes the length of a list (Example 1.9.11) may now be written as follows:

$$\mathtt{letrec}\,\mathtt{length} = \lambda(\mathtt{Nil}\ \_\,.\,\hat{0} \mid \mathtt{Cons}\ (\_, \mathtt{z})\,.\,\hat{1} \mathbin{\hat{+}} \mathtt{length}\ \mathtt{z})$$

The second pattern matches a nonempty list and binds `z` to its tail at the same time, obviating the need for an explicit application of π_2. □

1.9.16 Exercise [★★, Recommended, $\not\rightarrow$]: Under the above definition of `length`, consider an application of `length` to the list $\mathtt{Cons}(\hat{0}, \mathtt{Nil}())$. After eliminating the syntactic sugar, determine by which reduction sequence this expression reduces to a value. □

Before we can proceed and extend the type system to deal with the new `match` construct, we must make two mild extensions to the syntax and meaning of constraints. First, if σ is $\forall\bar{\mathtt{X}}[C].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}')$, then $\mathtt{T}' \preceq \sigma$ stands for the constraint $\exists\bar{\mathtt{X}}.(C \wedge \mathtt{T}' \leq \mathtt{T})$. This relation is identical to the instance relation (Definition 1.3.3), except the direction of subtyping is reversed. We extend the syntax of constraints with instantiation constraints of the form $\mathtt{T} \preceq \mathtt{x}$ and define their meaning by adding a symmetric counterpart of CM-Instance. We remark that, when subtyping is interpreted as equality, the relations $\sigma \preceq \mathtt{T}$ and $\mathtt{T} \preceq \sigma$ coincide, so this extension is unnecessary in that particular case. Second, we extend the syntax of environments so that several successive bindings may *share* a set of quantifiers and a constraint. That is, we allow writing $\forall\bar{\mathtt{X}}[C].(\mathtt{x}_1 : \mathtt{T}_1; \ldots; \mathtt{x}_k : \mathtt{T}_k)$ for $\mathtt{x}_1 : \forall\bar{\mathtt{X}}[C].\mathtt{T}_1; \ldots; \mathtt{x}_k : \forall\bar{\mathtt{X}}[C].\mathtt{T}_k$. From a theoretical standpoint, this is little more than syntactic sugar; however, in practice, it is useful to implement this new idiom literally, since it avoids unnecessary copying of the constraint C.

Let us now extend the type system. For the sake of brevity, we extend the constraint generation rules only. Of course, it would also be possible to define corresponding extensions of the rule-based type systems shown earlier, namely DM, HM(X), and PCB(X). We begin by defining a constraint $[\![\mathtt{T} : \mathtt{p}]\!]$ that represents a necessary and sufficient condition for values of type `T` to be acceptable inputs for the pattern `p`. Its free type variables are a subset of

$$
\begin{array}{rcl}
[\![\mathtt{T} : \_]\!] & = & \mathsf{true} \\
[\![\mathtt{T} : \mathtt{z}]\!] & = & \mathtt{T} \preceq \mathtt{z} \\
[\![\mathtt{T} : \mathtt{c}\,\mathtt{p}_1\ \ldots\ \mathtt{p}_k]\!] & = & \exists\bar{\mathtt{X}}.(\vec{\mathtt{X}} \rightarrow \mathtt{T} \preceq \mathtt{c} \wedge \bigwedge_{i=1}^{k} [\![\mathtt{X}_i : \mathtt{p}_i]\!]) \\
[\![\mathtt{T} : \mathtt{p}_1 \wedge \mathtt{p}_2]\!] & = & [\![\mathtt{T} : \mathtt{p}_1]\!] \wedge [\![\mathtt{T} : \mathtt{p}_2]\!] \\
[\![\mathtt{T} : \mathtt{p}_1 \vee \mathtt{p}_2]\!] & = & [\![\mathtt{T} : \mathtt{p}_1]\!] \wedge [\![\mathtt{T} : \mathtt{p}_2]\!]
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathtt{match}\ \mathtt{t}\ \mathtt{with}\ (\mathtt{p}_i\,.\,\mathtt{t}_i)_{i=1}^{k} : \mathtt{T}]\!] & = & \bigwedge_{i=1}^{k} \mathsf{let}\ \forall\mathtt{X}\bar{\mathtt{X}}_i[[\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]].(\vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i)\ \mathsf{in}\ [\![\mathtt{t}_i : \mathtt{T}]\!] \\
& & \text{where } \vec{\mathtt{z}}_i = dpi(\mathtt{p}_i)
\end{array}
$$

Figure 1-15: Constraint generation for patterns and pattern matching

ftv(T), while its free program identifiers are either constructors or program variables bound by p. It is defined in the upper part of Figure 1-15. The first rule states that a wildcard matches values of arbitrary type. The second and third rules govern program variables and constructor applications in *patterns*. They are identical to the rules that govern these constructs in *expressions* (page 59), except that the direction of subtyping is reversed. In the absence of subtyping, they would be entirely identical. We write $\vec{\mathtt{X}}$ for $\mathtt{X}_1 \ldots \mathtt{X}_k$ and $\vec{\mathtt{X}} \rightarrow \mathtt{T}$ for $\mathtt{X}_1 \rightarrow \ldots \rightarrow \mathtt{X}_k \rightarrow \mathtt{T}$. As usual, the type variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$ must have kind $\star$ and must be distinct and fresh for the equation's left-hand side. The last two rules simply distribute the type T to both subpatterns. It is easy to check that $[\![\mathtt{T} : \mathtt{p}]\!]$ is *contravariant* in T:

1.9.17 Lemma: $\mathtt{T}' \leq \mathtt{T} \wedge [\![\mathtt{T} : \mathtt{p}]\!]$ entails $[\![\mathtt{T}' : \mathtt{p}]\!]$. □

This property reflects the fact that T represents the type of an *input* for the pattern p. Compare it with Lemma 1.6.3.

1.9.18 Example: Consider the pattern $\mathtt{Cons}\,(\_, \mathtt{z})$, which appears in Example 1.9.15. We have

$$
\begin{array}{rl}
 & [\![\mathtt{T} : \mathtt{Cons}\,(\_, \mathtt{z})]\!] \\
\equiv & \exists\mathtt{Z}_1.([\![\mathtt{Z}_1 \rightarrow \mathtt{T} : \mathtt{Cons}]\!] \wedge [\![\mathtt{Z}_1 : (\_, \mathtt{z})]\!]) \\
\equiv & \exists\mathtt{Z}_1.(\mathtt{Z}_1 \rightarrow \mathtt{T} \preceq \mathtt{Cons} \wedge \exists\mathtt{Z}_2\mathtt{Z}_3.([\![\mathtt{Z}_2 \rightarrow \mathtt{Z}_3 \rightarrow \mathtt{Z}_1 : (\cdot,\cdot)]\!] \wedge [\![\mathtt{Z}_2 : \_]\!] \wedge [\![\mathtt{Z}_3 : \mathtt{z}]\!])) \\
\equiv & \exists\mathtt{Z}_1\mathtt{Z}_2\mathtt{Z}_3.(\mathtt{Z}_1 \rightarrow \mathtt{T} \preceq \mathtt{Cons} \wedge \mathtt{Z}_2 \rightarrow \mathtt{Z}_3 \rightarrow \mathtt{Z}_1 \preceq (\cdot,\cdot) \wedge \mathtt{Z}_3 \preceq \mathtt{z})
\end{array}
$$

where $\mathtt{Z}_1$, $\mathtt{Z}_2$, $\mathtt{Z}_3$ are fresh for T. Let us now place this constraint within the scope of the initial environment, which assigns type schemes to the constructors Cons and $(\cdot,\cdot)$, and within the scope of a binding of z to some type $\mathtt{T}'$.

We find

$$\begin{array}{rl} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \mathsf{let}\ \mathtt{z} : \mathtt{T}'\ \mathsf{in}\ [\![\mathtt{T} : \mathtt{Cons}\ (\_, \mathtt{z})]\!] \\ \equiv & \exists \mathtt{Z}_1\mathtt{Z}_2\mathtt{Z}_3.(\exists \mathtt{X}.(\mathtt{Z}_1 \to \mathtt{T} \leq \mathtt{X} \times \mathsf{list}\,\mathtt{X} \to \mathsf{list}\,\mathtt{X}) \wedge \\ & \qquad \exists \mathtt{Y}_1\mathtt{Y}_2.(\mathtt{Z}_2 \to \mathtt{Z}_3 \to \mathtt{Z}_1 \leq \mathtt{Y}_1 \to \mathtt{Y}_2 \to \mathtt{Y}_1 \times \mathtt{Y}_2) \wedge \mathtt{Z}_3 \leq \mathtt{T}') \\ \equiv & \exists \mathtt{X}.(\mathtt{T} \leq \mathsf{list}\,\mathtt{X} \wedge \mathsf{list}\,\mathtt{X} \leq \mathtt{T}') \end{array}$$

where the final simplification relies mainly on C-Arrow, on the corresponding rule for products, and on C-ExTrans, and is left as an exercise to the reader. Thus, the constraint states that the pattern matches values that have type $\mathsf{list}\,\mathtt{X}$ (equivalently, values whose type $\mathtt{T}$ is a subtype of $\mathsf{list}\,\mathtt{X}$), for some undetermined element type $\mathtt{X}$, and binds $\mathtt{z}$ to values of type $\mathsf{list}\,\mathtt{X}$ (equivalently, values whose type $\mathtt{T}'$ is a supertype of $\mathsf{list}\,\mathtt{X}$). $\square$

The above example seems to indicate that the constraint generation rules for patterns make some sense. Still, the careful reader may be somewhat puzzled by the third rule, which, compared to its analogue for expressions, reverses the direction of subtyping, but does not reverse the direction of instantiation. Indeed, in order for this rule to make sense, and to be sound, we must formulate a requirement concerning the type schemes assigned to constructors.

1.9.19 Definition: A constructor $\mathtt{c}$ is *invertible* if and only if, when $\vec{\mathtt{X}}$ and $\vec{\mathtt{X}}'$ have length $a(\mathtt{c})$, the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\vec{\mathtt{X}}' \to \mathtt{T} \preceq \mathtt{c} \wedge \mathtt{c} \preceq \vec{\mathtt{X}} \to \mathtt{T})$ entails $\vec{\mathtt{X}} \leq \vec{\mathtt{X}}'$. In the following, we assume patterns contain invertible constructors only. $\square$

Intuitively, when $\mathtt{c}$ is invertible, it is possible to recover the type of every $\mathtt{v}_i$ from the type of $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$, a crucial property for pattern matching to be possible. Please note that, if $\Gamma_0(\mathtt{c})$ is monomorphic, then $\mathtt{c}$ is invertible. The following lemma identifies another important class of invertible constructors.

1.9.20 Lemma: The constructors of algebraic data types are invertible. $\square$

Proof: Let $\mathtt{c}$ be a constructor introduced by the definition of an algebraic data type $\mathtt{D}$. Let $k = a(\mathtt{c})$. Then, the type scheme $\Gamma_0(\mathtt{c})$ is of the form $\forall\bar{\mathtt{Y}}.\vec{\mathtt{T}} \to \mathtt{D}\,\vec{\mathtt{Y}}$, where $\vec{\mathtt{Y}}$ are the parameters of the definition and $\vec{\mathtt{T}}$, a vector of length k, consists of *some of* the definition's components. (More precisely, $\vec{\mathtt{T}}$ contains just one component in the case of variant types and contains all components in the case of record types.) Let $\vec{\mathtt{X}}$ and $\vec{\mathtt{X}}'$ have length k. Let $\forall\bar{\mathtt{Y}}_1.\vec{\mathtt{T}}_1 \to \mathtt{D}\,\vec{\mathtt{Y}}_1$ and $\forall\bar{\mathtt{Y}}_2.\vec{\mathtt{T}}_2 \to \mathtt{D}\,\vec{\mathtt{Y}}_2$ be two α-equivalent forms of the type scheme $\Gamma_0(\mathtt{c})$, with $\bar{\mathtt{Y}}_1 \# \bar{\mathtt{Y}}_2$ and $\bar{\mathtt{Y}}_1\bar{\mathtt{Y}}_2 \# \mathit{ftv}(\bar{\mathtt{X}}, \bar{\mathtt{X}}', \mathtt{T})$. The constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\vec{\mathtt{X}}' \to \mathtt{T} \preceq \mathtt{c} \wedge \mathtt{c} \preceq \vec{\mathtt{X}} \to \mathtt{T})$ is, by definition, equivalent to $\vec{\mathtt{X}}' \to \mathtt{T} \preceq \Gamma_0(\mathtt{c}) \wedge \Gamma_0(\mathtt{c}) \preceq \vec{\mathtt{X}} \to \mathtt{T}$, that is, $\exists\bar{\mathtt{Y}}_1.(\vec{\mathtt{X}}' \to \mathtt{T} \leq \vec{\mathtt{T}}_1 \to \mathtt{D}\,\vec{\mathtt{Y}}_1) \wedge \exists\bar{\mathtt{Y}}_2.(\vec{\mathtt{T}}_2 \to \mathtt{D}\,\vec{\mathtt{Y}}_2 \leq \vec{\mathtt{X}} \to \mathtt{T})$. By C-ExAnd and C-Arrow, this may be written $\exists\bar{\mathtt{Y}}_1\bar{\mathtt{Y}}_2.(\mathtt{D}\,\vec{\mathtt{Y}}_2 \leq \mathtt{T} \leq \mathtt{D}\,\vec{\mathtt{Y}}_1 \wedge \vec{\mathtt{X}} \leq \vec{\mathtt{T}}_2 \wedge \vec{\mathtt{T}}_1 \leq \vec{\mathtt{X}}')$. Now, by Definition 1.9.8, $\mathtt{D}\,\vec{\mathtt{Y}}_2 \leq \mathtt{D}\,\vec{\mathtt{Y}}_1$ entails $\vec{\mathtt{T}}_2 \leq \vec{\mathtt{T}}_1$, so the previous constraint entails $\exists\bar{\mathtt{Y}}_1\bar{\mathtt{Y}}_2.(\vec{\mathtt{X}} \leq \vec{\mathtt{X}}')$, that is, $\vec{\mathtt{X}} \leq \vec{\mathtt{X}}'$. □

An important class of *noninvertible* constructors are those associated with existential type definitions (page 118), where not all quantifiers of the type scheme $\Gamma_0(\mathtt{c})$ are parameters of the type constructor $\mathtt{D}$. For instance, under the definition $\mathtt{D} \approx \ell : \exists\mathtt{X}.\mathtt{X}$, the type scheme associated with ℓ is $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{D}$. Then, it is easy to check that ℓ is not invertible. This reflects the fact that it is not possible to recover the type of $\mathtt{v}$ from the type of $\ell\ \mathtt{v}$—which must be $\mathtt{D}$ in any case—and explains why existential types require special treatment.

We are now ready to associate a constraint generation rule with the `match` construct. It is given in the lower part of Figure 1-15. In the rule's right-hand side, we write $\vec{\mathtt{z}}_i$ for the program variables bound by the pattern $\mathtt{p}_i$, and we write $\vec{\mathtt{X}}_i$ for a vector of type variables of the same length. The type variables $\mathtt{X}\bar{\mathtt{X}}_i$ must have kind $\star$, must be pairwise distinct and must not appear free in the rule's left-hand side. Let us now explain the rule. Its right-hand side is a conjunction, where each conjunct deals with one clause of the `match` construct, requiring $\mathtt{t}_i$ to have type $\mathtt{T}$ under certain assumptions about the program variables $\vec{\mathtt{z}}_i$ bound by the pattern $\mathtt{p}_i$. There remains to explain how these assumptions are built. First, as in the case of a `let` construct, we summon a fresh type variable $\mathtt{X}$ and produce $[\![\mathtt{t} : \mathtt{X}]\!]$, the least specific constraint that guarantees $\mathtt{t}$ has type $\mathtt{X}$. Then, reflecting the operational semantics, which feeds (the value produced by) $\mathtt{t}$ into the pattern $\mathtt{p}_i$, we feed the type $\mathtt{X}$ into $\mathtt{p}_i$ and produce $\mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]$, a constraint that guarantees that $\vec{\mathtt{X}}_i$ is a correct vector of type assumptions for the program variables $\vec{\mathtt{z}}_i$ (see Example 1.9.18). This explains why we may place $[\![\mathtt{T} : \mathtt{t}_i]\!]$ within the scope of $(\vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i)$. There remains to point out that, as in the case of the `let` construct, *every* assignment of ground types to $\mathtt{X}\bar{\mathtt{X}}_i$ that satisfies the constraint $[\![\mathtt{t} : \mathtt{X}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}_i\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]$ is acceptable, so it is valid to universally quantify these type variables. This allows the program variables $\vec{\mathtt{z}}_i$ to receive polymorphic type schemes when $\mathtt{t}$ itself has polymorphic type.

1.9.21 Exercise [★, Recommended]: We have previously suggested viewing `let z = t`$_1$ `in t`$_2$ as syntactic sugar for `match t`$_1$ `with z.t`$_2$, and shown that the operational semantics validates this view. Check that it is also valid from a typing perspective. □

The `match` constraint generation rule, if implemented literally, takes k copies of the constraint $[\![\mathtt{t} : \mathtt{X}]\!]$. When k is greater than 1, this compromises the linear time and space complexity of constraint generation. To remedy this problem, one may modify the rule as follows: replace every copy of $[\![\mathtt{t} : \mathtt{X}]\!]$ with $\mathtt{z} \preceq \mathtt{X}$ and place the constraint within the context $\mathsf{let}\ \mathtt{z} : \forall\mathtt{X}[[\![\mathtt{t} : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ []$, where $\mathtt{z}$ is a fresh program variable. It is not difficult to check that the logical meaning of the constraint is not affected and that a linear behavior is recovered. In practice, solving the new constraint requires taking instances of the type scheme $\forall \mathtt{X}[[\![\mathtt{t} : \mathtt{X}]\!]].\mathtt{X}$, which essentially requires copying $[\![\mathtt{t} : \mathtt{X}]\!]$ again—however, an efficient solver may now *simplify* this subconstraint before duplicating it.

The following lemma is a key to establishing subject reduction for R-Match. It relies on the requirement that constructors be invertible.

1.9.22 Lemma: Assume $[\mathtt{p} \mapsto \mathtt{v}]$ is defined and maps $\vec{\mathtt{z}}$ to $\vec{\mathtt{w}}$, where $\bar{\mathtt{z}} = dpi(\mathtt{p})$. Let $\vec{\mathtt{z}} : \vec{\mathtt{T}}$ be an arbitrary monomorphic environment of domain $\bar{\mathtt{z}}$. Then, $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{T}]\!] \wedge \mathsf{let}\ \vec{\mathtt{z}} : \vec{\mathtt{T}}\ \mathsf{in}\ [\![\mathtt{T} : \mathtt{p}]\!])$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\vec{\mathtt{w}} : \vec{\mathtt{T}}]\!]$. □

We now prove that our extension of ML-the-calculus with pattern matching enjoys subject reduction. We only state that R-Match preserves types, and leave the new subcase of R-Context, where the evaluation context involves a `match` construct, to the reader. For this subcase to succeed, the value restriction (Definition 1.7.7) must be extended to require that either all constants have pure semantics or all `match` constructs are in fact of the form $\mathtt{match\ v\ with}\ (\mathtt{p}_i . \mathtt{t}_i)_{i=1}^{k}$.

1.9.23 Theorem [Subject reduction]: (R-Match) $\subseteq$ ($\sqsubseteq$). □

1.9.24 Exercise [★★★, $\not\rightarrow$]: For the sake of simplicity, we have omitted the production `ref p` from the syntax of patterns. The pattern `ref p` matches every memory location whose content (with respect to the current store) is matched by `p`. Determine how the previous definitions and proofs must be extended in order to accommodate this new production. □

The progress property does *not* hold in general: for instance, $\mathtt{match\ Nil\ with}\ (\mathtt{Cons\ z} . \mathtt{z})$ is well-typed (with type $\forall \mathtt{X}.\mathtt{X}$) but is stuck. In actual implementations of ML-the-programming-language, such errors are dynamically detected. This may be considered a weakness of ML-the-type-system. Fortunately, however, it is often possible to statically prove that a particular `match` construct is *exhaustive* and cannot go wrong. Indeed, if $\mathtt{match\ v\ with}\ (\mathtt{p}_i . \mathtt{t}_i)_{i=1}^{k}$ is well-typed, then for every $i \in \{1, \ldots, k\}$, the constraint $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{X}]\!] \wedge \exists \bar{\mathtt{X}}.\mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!])$, where $\bar{\mathtt{z}}_i$ are the program variables bound by $\mathtt{p}_i$, must be satisfiable; that is, `v` must have some type that is an acceptable input for $\mathtt{p}_i$. This fact yields information about `v`, from which it may be possible to derive that `v` must match one of the patterns $\mathtt{p}_i$.

1.9.25 Example: Let $k = 2$, $\mathtt{p}_1 = \mathtt{Nil\ \_}$, and $\mathtt{p}_2 = \mathtt{Cons}\ (\mathtt{z}_1, \mathtt{z}_2)$. Then, the constraints $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{\mathtt{X}}.\mathsf{let}\ \vec{\mathtt{z}}_i : \vec{\mathtt{X}}\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{p}_i]\!]$, for $i \in \{1, 2\}$, are both equivalent (after simplification, when $i = 2$) to $\exists \mathtt{Z}.\mathtt{X} \leq \mathsf{list}\ \mathtt{Z}$. Because the type constructor list is isolated, every closed value `v` whose type `X` satisfies this constraint must be an application of $\mathtt{Nil}$ or $\mathtt{Cons}$. If the latter, because $\mathtt{Cons}$ has type $\forall \mathtt{X}.\mathtt{X} \times \mathsf{list}\,\mathtt{X} \to \mathsf{list}\,\mathtt{X}$, and because the type constructor $\times$ is isolated, the argument to $\mathtt{Cons}$ must be a pair. We conclude that $\mathtt{v}$ must match either $\mathtt{p}_1$ or $\mathtt{p}_2$, which guarantees that this `match` construct is exhaustive and its evaluation cannot go wrong. □

It is beyond the scope of this chapter to give more details about the check for exhaustiveness. The reader is referred to (Sekar, Ramesh, and Ramakrishnan, 1995; Le Fessant and Maranget, 2001).

Type annotations

So far, we have been interested in a very pure, and extreme, form of type inference. Indeed, in ML-the-calculus, expressions contain no explicit type information whatsoever: it is entirely inferred. In practice, however, it is often useful to insert *type annotations* within expressions, because they provide a form of machine-checked documentation. Type annotations are also helpful when attempting to trace the cause of a type error: by supplying the type-checker with (supposedly) correct type information, one runs a better chance of finding a type inconsistency near an actual programming mistake.

When type annotations are allowed to contain type variables, one must be quite careful about *where* (at which program point) and *how* (existentially or universally) these variables are bound. Indeed, the meaning of type annotations cannot be made precise without settling these issues. In what follows, we first explain how to introduce type annotations whose type variables are bound *locally* and existentially. We show that extending ML-the-calculus with such limited type annotations is again a simple matter of introducing new constants. Then, we turn to a more general case, where type variables may be explicitly existentially introduced *at any program point*. We defer the discussion of *universally* bound type variables to Section 1.10.

Let a *local existential type annotation* $\exists\bar{\mathtt{X}}.\mathtt{T}$ be a pair of a set of type variables $\bar{\mathtt{X}}$ and a type $\mathtt{T}$, where $\mathtt{T}$ has kind $\star$, $\bar{\mathtt{X}}$ is considered bound within $\mathtt{T}$, and $\bar{\mathtt{X}}$ contains $ftv(\mathtt{T})$. For every such annotation, we introduce a new unary destructor $(\cdot : \exists\bar{\mathtt{X}}.\mathtt{T})$. Such a definition is valid only because a type annotation must be *closed*, that is, does not have any free type variables. We write $(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T})$ for the application $((\cdot : \exists\bar{\mathtt{X}}.\mathtt{T}))\ \mathtt{t}$. Since a type annotation does not affect the meaning of a program, the new destructor has identity semantics:

$$(\mathtt{v} : \exists\bar{\mathtt{X}}.\mathtt{T}) \xrightarrow{\delta} \mathtt{v} \qquad \text{(R-Annotation)}$$

Its type scheme, however, is not that of the identity, namely $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$: instead, it is less general, so that annotating an expression *restricts* its type. Indeed, we extend the initial environment Γ_0 with the binding

$$(\cdot : \exists \bar{\mathtt{X}}.\mathtt{T}) : \forall \bar{\mathtt{X}}.\mathtt{T} \to \mathtt{T}$$

1.9.26 Exercise [★]: Check that $\forall \bar{\mathtt{X}}.\mathtt{T} \to \mathtt{T}$ is an *instance* of $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$ in Damas and Milner's sense, that is, the former is obtained from the latter via the rule DM-Inst' given in Exercise 1.2.23. Does this allow arguing that the type scheme assigned to $(\cdot : \exists \bar{\mathtt{X}}.\mathtt{T})$ is sound? Check that the above definitions meet the requirements of Definition 1.7.6. □

Although inserting a type annotation does not change the semantics of the program, it does affect constraint generation, hence type inference. We let the reader check that, assuming $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t}, \mathtt{T}')$, the following derived constraint generation rule holds:

$$\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\mathtt{t} : \exists \bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] \equiv \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \bar{\mathtt{X}}.([\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}')$$

So far, expressions cannot have free type variables, so the hypothesis $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$ may seem superfluous. However, we shall soon allow expressions to contain type annotations with free type variables, so we prefer to make this condition explicit now. According to this rule, the effect of the type annotation is to force the expression $\mathtt{t}$ to have type $\mathtt{T}$, for *some* choice of the type variables $\bar{\mathtt{X}}$. As usual in type systems with subtyping, the expression's final type $\mathtt{T}'$ may then be an arbitrary supertype of this particular instance of $\mathtt{T}$. When subtyping is interpreted as equality, $\mathtt{T}'$ and $\mathtt{T}$ are equated by the constraint, so this constraint generation rule may be read: *a valid type for* $(\mathtt{t} : \exists \bar{\mathtt{X}}.\mathtt{T})$ *must be of the form* $\mathtt{T}$*, for some choice of the type variables* $\bar{\mathtt{X}}$*.*

1.9.27 Example: In DM extended with integers, the expression $(\lambda \mathtt{z}.\mathtt{z} : \mathtt{int} \to \mathtt{int})$ has most general type $\mathtt{int} \to \mathtt{int}$, even though the underlying identity function has most general type $\forall \mathtt{X}.\mathtt{X} \to \mathtt{X}$, so the annotation restricts its type. The expression $(\lambda \mathtt{z}.\mathtt{z} \hat{+} \hat{1} : \exists \mathtt{X}.\mathtt{X} \to \mathtt{X})$ has type $\mathtt{int} \to \mathtt{int}$, which is also the most general type of the underlying function, so the annotation acts merely as documentation in this case. Note that the type variable $\mathtt{X}$ is instantiated to $\mathtt{int}$ by the constraint solver. The expression $(\lambda \mathtt{z}.(\mathtt{z}\ \hat{1}) : \exists \mathtt{X}.\mathtt{X} \to \mathtt{int})$ has type $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$ because the underlying function has type $(\mathtt{int} \to \mathtt{Y}) \to \mathtt{Y}$, which successfully unifies with $\mathtt{X} \to \mathtt{int}$ by instantiating $\mathtt{X}$ to $\mathtt{int} \to \mathtt{int}$ and $\mathtt{Y}$ to $\mathtt{int}$. Last, the expression $(\lambda \mathtt{z}.(\mathtt{z}\ \hat{1}) : \exists \mathtt{X}.\mathtt{int} \to \mathtt{X})$ is ill-typed—even though the underlying expression is well-typed—because the equation $(\mathtt{int} \to \mathtt{Y}) \to \mathtt{Y} = \mathtt{int} \to \mathtt{X}$ is unsatisfiable. □

1.9.28 EXAMPLE: In DM extended with pairs, the expression $\lambda z_1.\lambda z_2.((z_1 : \exists X.X), (z_2 : \exists X.X))$ has most general type $\forall XY.X \to Y \to X \times Y$. In other words, the two occurrences of X do not represent the same type. Indeed, one could just as well have written $\lambda z_1.\lambda z_2.((z_1 : \exists X.X), (z_2 : \exists Y.Y))$. If one wishes z_1 and z_2 to receive the same type, one must lift the type annotations and merge them above the pair constructor, as follows: $\lambda z_1.\lambda z_2.((z_1, z_2) : \exists X.X \times X)$. In the process, the type constructor $\times$ has appeared in the annotation, causing its size to increase. □

The above example reveals a limitation of this style of type annotations: by requiring every type annotation to be closed, we lose the ability for two separate annotations to *share* a type variable. Yet, such a feature is sometimes desirable. If the two annotations where sharing is desired are distant in the code, it may be awkward to lift and merge them into a single annotation; so, more expressive power is sometimes truly needed.

Thus, we are lead to consider more general type annotations, of the form $(t : T)$, where T has kind $\star$, and where the type variables that appear within T are considered *free*, so that distinct type annotations may refer to shared type variables. For this idea to make sense, however, it is still necessary to specify where these type variables are bound. We do so using expressions of the form $\exists \bar{X}.t$. Such an expression binds the type variables $\bar{X}$ within the expression t, so that all free occurrences of X (where $X \in \bar{X}$) in type annotations inside t stand for the same type. Thus, we break the simple type annotation construct $(\cdot : \exists \bar{X}.T)$ into two more elementary constituents, namely *existential type variable introduction* $\exists \bar{X}.\cdot$ and *type constraint* $(\cdot : T)$. Note that both are new forms of expressions; neither can be encoded by adding new constants to the calculus, because it is not possible to assign *closed* type schemes to them.

Technically, allowing expressions to contain type variables requires some care. Several constraint generation rules employ auxiliary type variables, which become bound in the generated constraint. These type variables may be chosen in an arbitrary way, provided they do not appear free in the rule's left-hand side—a side-condition intended to avoid inadvertent capture. So far, this side-condition could be read: the auxiliary type variables used to form the constraint $[\![t : T]\!]$ must not appear free within T. Now, since type annotations may contain free type variables, the side-condition becomes: *the auxiliary type variables used to form* $[\![t : T]\!]$ *must not appear free within* t *or* T.

With this extended side-condition in mind, our original constraint generation rules remain unchanged. We add two new rules to describe how the new expression forms affect constraint generation:

$$\begin{aligned} [\![\exists \bar{X}.t : T]\!] &= \exists \bar{X}.[\![t : T]\!] \qquad \text{provided } \bar{X} \# \mathit{ftv}(T) \\ [\![(t : T) : T']\!] &= [\![t : T]\!] \wedge T \leq T' \end{aligned}$$

The effect of these rules is simple. The construct $\exists\bar{\mathtt{X}}.\mathtt{t}$ is an indication to the constraint generator that the type variables $\bar{\mathtt{X}}$, which may occur free within type annotations inside $\mathtt{t}$, should be existentially bound at this point. The side-condition $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$ ensures that quantifying over $\bar{\mathtt{X}}$ in the generated constraint does not capture type variables in the expected type $\mathtt{T}$. It can always be satisfied by α-conversion of the expression $\exists\bar{\mathtt{X}}.\mathtt{t}$. The construct $(\mathtt{t} : \mathtt{T})$ is an indication to the constraint generator that the expression $\mathtt{t}$ should have type $\mathtt{T}$, and it is treated as such by generating the subconstraint $[\![\mathtt{t} : \mathtt{T}]\!]$. The expression's type may be an arbitrary supertype of $\mathtt{T}$, hence the auxiliary constraint $\mathtt{T} \leq \mathtt{T}'$.

1.9.29 Example: In DM extended with pairs, the expression $\lambda\mathtt{z}_1.\lambda\mathtt{z}_2.\exists\mathtt{X}.((\mathtt{z}_1 : \mathtt{X}), (\mathtt{z}_2 : \mathtt{X}))$ has most general type $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X} \rightarrow \mathtt{X} \times \mathtt{X}$. Indeed, the constraint generated for this expression contains the pattern $\exists\mathtt{X}.([\![\mathtt{z}_1 : \mathtt{X}]\!] \wedge [\![\mathtt{z}_2 : \mathtt{X}]\!] \wedge \ldots)$, which causes $\mathtt{z}_1$ and $\mathtt{z}_2$ to receive the same type. Note that this style is more flexible than that employed in Example 1.9.28, where we were forced to use a single, monolithic type annotation to express this sharing constraint. □

1.9.30 Remark: In practice, a type variable is usually represented as a memory cell in the typechecker's heap. So, one cannot say that the source code contains type variables; rather, it contains *names* that are meant to stand for type variables. Let us write X for such a name, and T for a type made of type constructors and names, rather than of type constructors and type variables. Then, our new expression forms are really $\exists\bar{X}.\mathtt{t}$ and $(\mathtt{t} : T)$. When the constraint generator enters the scope of an introduction form $\exists\bar{X}.\mathtt{t}$, it allocates a vector of fresh type variables $\bar{\mathtt{X}}$, and augments an internal environment with the bindings $\bar{X} \mapsto \bar{\mathtt{X}}$. Because the type variables are fresh, the side-condition of the first constraint generation rule above is automatically satisfied. When the constraint generator finds a type annotation $(\mathtt{t} : T)$, it looks up the internal environment to translate the type annotation T into an internal type $\mathtt{T}$—which fails if T contains a name that is not in scope—and applies the second constraint generation rule above. □

1.9.31 Exercise [★★, ↛]: Let $\bar{\mathtt{X}} \supseteq \mathit{ftv}(\mathtt{T})$ and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$. Check that the constraints $[\![(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!]$ and $[\![\exists\bar{\mathtt{X}}.(\mathtt{t} : \mathtt{T}) : \mathtt{T}']\!]$ are equivalent. In other words, the *local* type annotations introduced earlier may be expressed in terms of the more complex constructs described above. □

1.9.32 Exercise [★★, ↛]: One way of giving identity semantics to our new type annotation constructs is to *erase* them altogether prior to execution. Give an inductive definition of $\lfloor\mathtt{t}\rfloor$, the expression obtained by removing all type annotation constructs from the expression $\mathtt{t}$. Check that $[\![\mathtt{t} : \mathtt{T}]\!]$ entails $[\![\lfloor\mathtt{t}\rfloor : \mathtt{T}]\!]$ and explain why this is sufficient to ensure type soundness. □

It is interesting to study how explicit introduction of existentially quantified type variables interacts with `let`-polymorphism. The source of their interaction lies in the difference between the constraints $\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}[\exists\mathtt{X}.C_1].\mathtt{T}\ \mathsf{in}\ C_2$ and $\exists\mathtt{X}.\mathsf{let}\ \mathtt{z} : \forall\bar{\mathtt{X}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2$, which was explained in Example 1.3.28. In the former constraint, every free occurrence of `z` inside C_2 causes a copy of $\exists\mathtt{X}.C_1$ to be taken, thus creating its own fresh copy of `X`. In the latter constraint, on the other hand, every free occurrence of `z` inside C_2 produces a copy of C_1. All such copies *share* references to `X`, because its quantifier was not duplicated. In the former case, one may say that the type scheme assigned to `z` is *polymorphic* with respect to `X`, while in the latter case it is *monomorphic*. As a result, the placement of type variable introduction expressions with respect to `let` bindings in the source code is meaningful: introducing a type variable *outside* of a `let` construct *prevents* it from being generalized.

1.9.33 Example: In DM extended with integers and Booleans, the program $\mathtt{let}\ \mathtt{f} = \exists\mathtt{X}.\lambda\mathtt{z}.(\mathtt{z} : \mathtt{X})\ \mathtt{in}\ (\mathtt{f}\ 0, \mathtt{f}\ \mathtt{true})$ is well-typed. Indeed, the type scheme assigned to `f` is $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$. However, the program $\exists\mathtt{X}.\mathtt{let}\ \mathtt{f} = \lambda\mathtt{z}.(\mathtt{z} : \mathtt{X})\ \mathtt{in}\ (\mathtt{f}\ 0, \mathtt{f}\ \mathtt{true})$ is ill-typed. Indeed, the type scheme assigned to `f` is $\mathtt{X} \rightarrow \mathtt{X}$; then, no value of `X` satisfies the constraints associated with the applications `f` 0 and `f true`. The latter behavior is observed in Objective Caml, where type variables are *implicitly* introduced at the outermost level of expressions:

```
# let f z = (z:'a) in (f 0, f true);;
This expression has type bool but is here used with type int
```

More details about the treatment of type annotations in Standard ML, Objective Caml, and Haskell are given on page 113. □

1.9.34 Exercise [★, ↛]: Determine which constraints are generated for the two programs in Example 1.9.33. Check that the former is indeed well-typed, while the latter is ill-typed. □

Recursive types

We have shown that specializing HM(X) with an equality-only syntactic model yields HM(=), a constraint-based formulation of Damas and Milner's type system. Similarly, it is possible to specialize HM(X) with an equality-only free *regular* tree model, yielding a constraint-based type system that may be viewed as an extension of Damas and Milner's type discipline with recursive types. This flavor of recursive types is sometimes known as *equirecursive*, since cyclic *equations*, such as $\mathtt{X} = \mathtt{X} \rightarrow \mathtt{X}$, are then satisfiable. Our theorems about type inference and type soundness, which are independent of the model, remain valid. The constraint solver described in Section 1.8 may be used in the setting of an equality-only free regular tree model: the only difference with the syntactic case is that the occurs check is no longer performed.

Please note that, although *ground types* are regular, *types* remain finite objects: their syntax is unchanged. The μ notation commonly employed to describe recursive types may be emulated using type equations: for instance, the notation $\mu \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$ corresponds, in our constraint-based approach, to the type scheme $\forall \mathtt{X}[\mathtt{X} = \mathtt{X} \rightarrow \mathtt{X}].\mathtt{X}$.

Although recursive types come for free, as explained above, they have not been adopted in mainstream programming languages based on ML-the-type-system. The reason is pragmatic: experience shows that many nonsensical expressions are well-typed in the presence of recursive types, whereas they are not in their absence. Thus, the gain in expressiveness is offset by the fact that many programming mistakes are detected later than otherwise possible. Consider, for instance, the following `OCaml` session:

```
ocaml -rectypes
# let rec map f = function
    | [] → []
    | x :: l → (map f x) :: (map f l);;
val map : 'a → ('b list as 'b) → ('c list as 'c) = <fun>
```

This nonsensical variant of `map` is essentially useless, yet well-typed. Its principal type scheme, in our notation, is $\forall \mathtt{XYZ}[\mathtt{Y} = \mathsf{list}\,\mathtt{Y} \wedge \mathtt{Z} = \mathsf{list}\,\mathtt{Z}].\mathtt{X} \rightarrow \mathtt{Y} \rightarrow \mathtt{Z}$. In the absence of recursive types, it is ill-typed, since the constraint $\mathtt{Y} = \mathsf{list}\,\mathtt{Y} \wedge \mathtt{Z} = \mathsf{list}\,\mathtt{Z}$ is then false.

The need for equirecursive types is usually suppressed by the presence of algebraic data types, which offer *isorecursive* types, in the language. Yet, they are still necessary in some situations, such as in Objective Caml's object-oriented extension (Rémy and Vouillon, 1998), where recursive object types are commonly inferred. In order to allow recursive object types while still rejecting the above variant of `map`, Objective Caml's constraint solver implements a selective occurs check, which forbids cycles unless they involve the type constructor $\langle \cdot \rangle$ associated with objects. The corresponding model is a tree model where every infinite path down a tree must encounter the type constructor $\langle \cdot \rangle$ infinitely often.

1.10 Universal quantification in constraints

The constraint logic studied so far allows a set of variables $\bar{\mathtt{X}}$ to be existentially quantified within a formula C. The resulting formula $\exists \bar{\mathtt{X}}.C$ receives its standard meaning: it requires C to hold for *some* $\bar{\mathtt{X}}$. However, we currently have no way of requiring a formula C to hold for *all* $\bar{\mathtt{X}}$. Is it possible to extend our logic with universal quantification? If so, what are the new possibilities offered by this extension, in terms of type inference? The present section proposes some answers to these questions.

It is worth noting that, although the standard notation for type schemes involves the symbol $\forall$, type scheme introduction and instantiation constraints do not allow an encoding of universal quantification. Indeed, a universal quantifier in a type scheme is very much like an existential quantifier in a constraint: this is suggested, for instance, by Definition 1.3.3 and by C-LetEx.

Constraints

We extend the syntax of constraints as follows:

$$C ::= \ldots \mid \forall \bar{\mathtt{X}}.C$$

Universally quantified variables are often referred to as *rigid*, while existentially quantified variables are known as *flexible*. The logical interpretation of constraints (Figure 1-5) is extended as follows:

$$\frac{\forall \vec{t} \quad \phi[\vec{\mathtt{X}} \mapsto \vec{t}] \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ C \qquad \bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)}{\phi \vdash \mathsf{def}\ \Gamma\ \mathsf{in}\ \forall \bar{\mathtt{X}}.C} \qquad \text{(CM-Forall)}$$

We let the reader check that none of the results established in Section 1.3 are affected by this addition. Furthermore, the extended constraint language enjoys the following properties.

1.10.1 Lemma: $\forall \bar{\mathtt{X}}.C \Vdash C$. Conversely, $\bar{\mathtt{X}} \# \mathit{ftv}(C)$ implies $C \Vdash \forall \bar{\mathtt{X}}.C$. □

1.10.2 Lemma: $\bar{\mathtt{X}} \# \mathit{ftv}(C_2)$ implies $\forall \bar{\mathtt{X}}.(C_1 \wedge C_2) \equiv (\forall \bar{\mathtt{X}}.C_1) \wedge C_2$. □

1.10.3 Lemma: $\forall \bar{\mathtt{X}}.\forall \bar{\mathtt{Y}}.C \equiv \forall \bar{\mathtt{X}}\bar{\mathtt{Y}}.C$. □

1.10.4 Lemma: Let $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$. Then, $\exists \bar{\mathtt{X}}.\forall \bar{\mathtt{Y}}.C$ entails $\forall \bar{\mathtt{Y}}.\exists \bar{\mathtt{X}}.C$. Conversely, if $\exists \bar{\mathtt{Y}}.C$ determines $\bar{\mathtt{X}}$, then $\forall \bar{\mathtt{Y}}.\exists \bar{\mathtt{X}}.C$ entails $\exists \bar{\mathtt{X}}.\forall \bar{\mathtt{Y}}.C$. □

Constraint solving

We briefly explain how to extend the constraint solver described in Section 1.8 with support for universal quantification. (Thus, we again assume an equality-only free tree model.) Constraint solving in the presence of equations and of existential and universal quantifiers is known as *unification under a mixed prefix*. It is a particular case of the decision problem for the first-order theory of equality on trees; see *e.g.* (Comon and Lescanne, 1989). Extending our solver is straightforward: in fact, the treatment of universal quantifiers turns

$$S; U; \forall \bar{\mathtt{X}}.C \quad \rightarrow \quad S[\forall \bar{\mathtt{X}}.[]]; U; C \qquad \text{(S-Solve-All)}$$
$$\text{if } \bar{\mathtt{X}} \# ftv(U)$$

$$S[\forall \bar{\mathtt{X}}.\exists \bar{\mathtt{Y}}\bar{\mathtt{Z}}.[]]; U; \mathsf{true} \quad \rightarrow \quad S[\exists \bar{\mathtt{Y}}.\forall \bar{\mathtt{X}}.\exists \bar{\mathtt{Z}}.[]]; U; \mathsf{true} \qquad \text{(S-AllEx)}$$
$$\text{if } \bar{\mathtt{X}} \# \bar{\mathtt{Y}} \wedge \exists \bar{\mathtt{X}}\bar{\mathtt{Z}}.U \text{ determines } \bar{\mathtt{Y}}$$

$$S[\forall \bar{\mathtt{X}}\mathtt{X}.\exists \bar{\mathtt{Y}}.[]]; U; \mathsf{true} \quad \rightarrow \quad \mathsf{false} \qquad \text{(S-All-Fail-1)}$$
$$\text{if } \mathtt{X} \notin \bar{\mathtt{Y}} \wedge \mathtt{X} \prec_U^\star \mathtt{Z} \wedge \mathtt{Z} \notin \mathtt{X}\bar{\mathtt{Y}}$$

$$S[\forall \bar{\mathtt{X}}\mathtt{X}.\exists \bar{\mathtt{Y}}.[]]; \mathtt{X} = \mathtt{T} = \epsilon \wedge U; \mathsf{true} \quad \rightarrow \quad \mathsf{false} \qquad \text{(S-All-Fail-2)}$$
$$\text{if } \mathtt{X} \notin \bar{\mathtt{Y}} \wedge \mathtt{T} \notin \mathcal{V}$$

$$S[\forall \bar{\mathtt{X}}.\exists \bar{\mathtt{Y}}.[]]; U_1 \wedge U_2; \mathsf{true} \quad \rightarrow \quad S; U_1; \mathsf{true} \qquad \text{(S-Pop-All)}$$
$$\text{if } \bar{\mathtt{X}}\bar{\mathtt{Y}} \# ftv(U_1) \wedge \exists \bar{\mathtt{Y}}.U_2 \equiv \mathsf{true}$$

Figure 1-16: Solving universal constraints

out to be surprisingly analogous to that of let constraints. To begin, we extend the syntax of stacks with so-called *universal frames*:

$$S ::= \ldots \mid S[\forall \bar{\mathtt{X}}.[]]$$

Because existential quantifiers cannot, in general, be hoisted out of universal quantifiers, rules S-Ex-1 to S-Ex-4 now allow floating them up to the nearest enclosing let *or universal* frame, if any, or to the outermost level, otherwise. Thus, in our machine representation of stacks, where rules S-Ex-1 to S-Ex-4 are applied in an eager fashion, every universal frame carries a list of the type variables that are existentially bound immediately after it, and integer ranks count not only let frames, but also universal frames.

The solver's specification is extended with the rules in Figure 1-16. S-Solve-All, a forward rule, discovers a universal constraint and enters it, creating a new universal frame to record its existence. S-AllEx exploits Lemma 1.10.4 to hoist existential quantifiers out of the universal frame. It is analogous to S-LetAll, and its implementation may rely upon the same procedure (Exercise 1.8.8). The next two rules detect failure conditions. S-All-Fail-1 states that the constraint $\forall \mathtt{X}.\exists \bar{\mathtt{Y}}.U$ is false if the rigid variable $\mathtt{X}$ is directly or indirectly *dominated* by a *free* variable $\mathtt{Z}$. Indeed, the value of $\mathtt{X}$ is then determined by that of $\mathtt{Z}$—but a universally quantified variable ranges over *all* values, so this is a contradiction. In such a case, $\mathtt{X}$ is commonly said to *escape its scope*. S-All-Fail-2 states that the same constraint is false if $\mathtt{X}$ is equated with a *nonvariable* term. Indeed, the value of $\mathtt{X}$ is then partially determined, since its head constructor is known, which again contradicts its universal status. Last, S-Pop-All splits the current unification constraint into two components U_1 and U_2, where U_1 is made up entirely of *old* variables and U_2 constrains *young* variables only. This decomposition is analogous to that performed by S-Pop-Let. Then, it is not difficult to check that $\forall\bar{\mathtt{X}}.\exists\bar{\mathtt{Y}}.(U_1 \wedge U_2)$ is equivalent to U_1. So, the universal frame, as well as U_2, are discarded, and the solver proceeds by examining whatever remains on top of the stack S.

It is possible to further extend the treatment of universal frames with two rules analogous to S-Compress and S-UnName. In practice, this improves the solver's efficiency, and makes it easier to share code between the treatment of let frames and that of universal frames.

It is interesting to remark that, as far as the underlying unification algorithm is concerned, there is no difference between existentially and universally quantified type variables. The algorithm solves whatever equations are presented to it, without inquiring about the status of their variables. Equations that lead to failure, because a rigid variable escapes its scope or is equated with a nonvariable term, are detected only when the universal frame is exited. A perhaps more common approach is to *mark* rigid variables as such, allowing the unification algorithm to signal failure as soon as one of the two error conditions is encountered. In this approach, a rigid variable may successfully unify only with itself or with flexible variables fresher than itself. It is often called a *Skolem constructor* in the literature (Läufer and Odersky, 1994; Shields and Peyton Jones, 2002). An interesting variant of this approach appears in Dowek, Hardin, Kirchner and Pfenning's treatment of (higher-order) unification (1995; 1998), where flexible variables are represented as ordinary variables, while rigid variables are encoded using De Bruijn indices.

The properties of our constraint solver are preserved by this extension: it is possible to prove that Lemmas 1.8.9, 1.8.10, and 1.8.11 remain valid.

Type annotations, continued

In Section 1.9, we introduced the expression form $(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T})$, allowing an expression $\mathtt{t}$ to be annotated with a type $\mathtt{T}$ whose free variables $\bar{\mathtt{X}}$ are *locally* and *existentially* bound. It is now natural to introduce the symmetric expression form $(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T})$, where $\mathtt{T}$ has kind $\star$, $\bar{\mathtt{X}}$ is bound within $\mathtt{T}$, and $\bar{\mathtt{X}}$ contains $ftv(\mathtt{T})$, as before. Its constraint generation rule is as follows:

$$[\![(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] \quad = \quad \forall\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \wedge \exists\bar{\mathtt{X}}.(\mathtt{T} \leq \mathtt{T}') \quad \text{provided } \bar{\mathtt{X}} \# ftv(\mathtt{t}, \mathtt{T}')$$

The first conjunct requires $\mathtt{t}$ to have type $\mathtt{T}$ for *all* values of $\bar{\mathtt{X}}$. Here, the type variables $\bar{\mathtt{X}}$ are *universally* bound, as expected. The second conjunct requires

$\mathtt{T}'$ to be *some* instance of the universal annotation $\forall\bar{\mathtt{X}}.\mathtt{T}$. Since $\mathtt{T}'$ is only a monotype, it seems difficult to think of another sensible way of constraining $\mathtt{T}'$. For this reason, the type variables $\bar{\mathtt{X}}$ are still *existentially* bound in the second conjunct. This makes the interpretation of the universal quantifier in type annotations a bit more complex than that of the existential quantifier. For instance, when subtyping is interpreted as equality, the constraint generation rule may be read: *a valid type for* $(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T})$ *is of the form* $\mathtt{T}$*, for* some *choice of the type variables* $\bar{\mathtt{X}}$*, provided* $\mathtt{t}$ *has type* $\mathtt{T}$ *for* all *choices of* $\bar{\mathtt{X}}$*.*

We remark that $(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T})$ must be a new expression form: it cannot be encoded by adding new constants to the calculus—whereas $(\mathtt{t} : \exists\bar{\mathtt{X}}.\mathtt{T})$ could—because none of the existing constraint generation rules produce universally quantified constraints. Like all type annotations, it has identity semantics.

What is the use of universal type annotations, compared with existential type annotations? When a type variable is existentially bound, the typechecker is free to assign it whatever value makes the program well-typed. As a result, the expressions $(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \exists\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$ and $(\lambda \mathtt{z}.\mathtt{z} : \exists\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$ are both well-typed: $\mathtt{X}$ is assigned $\mathtt{int}$ in the former case, and remains undetermined in the latter. However, it is sometimes useful to be able to insist that an expression should be polymorphic. This effect is naturally achieved by using a universally bound type variable. Indeed, $(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$ is ill-typed, because $\forall\mathtt{X}.(\mathtt{X} = \mathtt{int})$ is false, while $(\lambda \mathtt{z}.\mathtt{z} : \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$ is well-typed.

1.10.5 Exercise [★]: Write down the constraints $\exists\mathtt{Z}.[\![(\lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!]$ and $\exists\mathtt{Z}.[\![(\lambda \mathtt{z}.\mathtt{z} : \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!]$, which tell whether these expressions are well-typed. Check that the former is false, while the latter is satisfiable. □

A universal type annotation, as defined above, is nothing but a (closed) Damas-Milner type scheme. Thus, the new construct $(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T})$ gives us the ability to ensure that the expression $\mathtt{t}$ admits the type scheme $\forall\bar{\mathtt{X}}.\mathtt{T}$. This feature is exploited at the module level in ML-the-programming-language, where it is necessary to check that the inferred type for a module component $\mathtt{t}$ is more general than the type scheme $\mathtt{S}$ that appears in the module's signature. In our view, this process simply consists in ensuring that $(\mathtt{t} : \mathtt{S})$ is well-typed.

In Section 1.9, we have pointed out that local (that is, closed) type annotations offer limited expressiveness, because they cannot share type variables. To lift this limitation, we have introduced the expression forms $\exists\bar{\mathtt{X}}.\mathtt{t}$ and $(\mathtt{t} : \mathtt{T})$. The former binds the type variables $\bar{\mathtt{X}}$ within $\mathtt{t}$, making them available for use in type annotations, and instructs the constraint generator to existentially quantify them at this point. The latter requires $\mathtt{t}$ to have $\mathtt{T}$. It is natural to proceed in the same manner in the case of universal type annotations. We now introduce the expression form $\forall\bar{\mathtt{X}}.\mathtt{t}$, which also binds $\bar{\mathtt{X}}$ within $\mathtt{t}$, but comes with a different constraint generation rule:

$$[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!] \quad = \quad \forall\bar{\mathtt{X}}.\exists\mathtt{Z}.[\![\mathtt{t} : \mathtt{Z}]\!] \wedge \exists\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \quad \text{provided } \bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T}) \wedge \mathtt{Z} \notin \mathit{ftv}(\mathtt{t})$$

This rule is a bit more complex than that associated with the expression form $\exists\bar{\mathtt{X}}.\mathtt{t}$. Again, this is due to the fact that we do not wish to overconstrain $\mathtt{T}$. The first exercise below shows that a more naïve version of the rule does not yield the desired behavior. The second exercise shows that this version does. The third exercise clarifies an efficiency concern.

1.10.6 Exercise [★]: Assume that $[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!]$ is defined as $\forall\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!]$, provided $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{T})$. Write down the constraint $[\![\forall\mathtt{X}.(\lambda\mathtt{z}.\mathtt{z} : \mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!]$. Can you describe its solutions? Does it have the intended meaning? □

1.10.7 Exercise [★★]: Let $\bar{\mathtt{X}} \supseteq \mathit{ftv}(\mathtt{T})$ and $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$. Check that the constraints $[\![(\mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!]$ and $[\![\forall\bar{\mathtt{X}}.(\mathtt{t} : \mathtt{T}) : \mathtt{T}']\!]$ are equivalent. In other words, *local* universal type annotations may also be expressed in terms of the more complex constructs described above. □

1.10.8 Exercise [★★★★, ↛]: The constraint generation rule that appears above compromises the linear time and space complexity of constraint generation, because it duplicates the term $\mathtt{t}$. It is possible to avoid this problem, but this requires a slight generalization of the constraint language. Let us write $\mathsf{let}\ \mathtt{x} : \forall\underline{\bar{\mathtt{X}}}\bar{\mathtt{Y}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2$ for $\forall\bar{\mathtt{X}}.\exists\bar{\mathtt{Y}}.C_1 \wedge \mathsf{def}\ \mathtt{x} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}[C_1].\mathtt{T}\ \mathsf{in}\ C_2$. In this extended let form, the underlined variables $\bar{\mathtt{X}}$ are interpreted as *rigid*, instead of *flexible*, while checking that C_1 is satisfiable. However, the type scheme associated with $\mathtt{x}$ is not affected. *Check that the above constraint generation rule may now be written as follows:*

$$[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!] \quad = \quad \mathsf{let}\ \mathtt{x} : \forall\underline{\bar{\mathtt{X}}}\mathtt{Z}[[\![\mathtt{t} : \mathtt{Z}]\!]].\mathtt{Z}\ \mathsf{in}\ \mathtt{x} \preceq \mathtt{T} \quad \text{provided } \mathtt{Z} \notin \mathit{ftv}(\mathtt{t})$$

Roughly speaking, the new rule forms a most general type scheme for $\mathtt{t}$, ensures that the type variables $\bar{\mathtt{X}}$ are unconstrained in it, and checks that $\mathtt{T}$ is an instance of it. Furthermore, it does not duplicate $\mathtt{t}$. To complete the exercise, *extend the specification of the constraint solver (Figures 1-12 and 1-16), as well as its implementation,* to deal with this extension of the constraint language. □

To conclude, let us once again stress that, if $\mathtt{T}$ has free type variables, the effect of the type annotation $(\mathtt{t} : \mathtt{T})$ depends on *how* and *where* they are bound. The effect of *how* stems from the fact that binding a type variable universally, rather than existentially, leads to a stricter constraint. Indeed, we let the reader check that $[\![\forall\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!]$ entails $[\![\exists\bar{\mathtt{X}}.\mathtt{t} : \mathtt{T}]\!]$, while the converse does not hold in general. The effect of *where* has been illustrated, in the case of existentially bound type variables, in Section 1.9. It is due, in that case, to the fact that let and $\exists$ do not commute. In the case of universally bound type variables, it may be imputed to the fact that $\forall$ and $\exists$ do not commute. For instance, $\lambda\mathtt{z}.\forall\mathtt{X}.(\mathtt{z} : \mathtt{X})$ is ill-typed, because *inside the λ-abstraction*, the program variable `z` cannot be said to have every type. However, $\forall\mathtt{X}.\lambda\mathtt{z}.(\mathtt{z} : \mathtt{X})$ is well-typed, because the identity function does have type $\mathtt{X} \rightarrow \mathtt{X}$ for every `X`.

1.10.9 Exercise [★]: Write down the constraints $\exists\mathtt{Z}.[\![\lambda\mathtt{z}.\forall\mathtt{X}.(\mathtt{z} : \mathtt{X}) : \mathtt{Z}]\!]$ and $\exists\mathtt{Z}.[\![\forall\mathtt{X}.\lambda\mathtt{z}.(\mathtt{z} : \mathtt{X}) : \mathtt{Z}]\!]$, which tell whether these expressions are well-typed. Is the former satisfiable? Is the latter? □

In Standard ML and Objective Caml, the type variables that appear in type annotations are *implicitly* bound. That is, there is no syntax in the language for the constructs $\exists\bar{\mathtt{X}}.\mathtt{t}$ and $\forall\bar{\mathtt{X}}.\mathtt{t}$. When a type annotation $(\mathtt{t} : \mathtt{T})$ contains a free type variable `X`, a fixed convention tells how and where `X` is bound. In Standard ML, `X` is *universally* bound at the nearest `val` binding that encloses all related occurrences of `X` (Milner, Tofte, and Harper, 1990). The 1997 revision of Standard ML (Milner, Tofte, Harper, and MacQueen, 1997b) slightly improves on this situation by allowing type variables to be *explicitly* introduced at `val` bindings. However, they still must be universally bound. In Objective Caml, `X` is *existentially* bound at the nearest enclosing toplevel `let` binding; this behavior seems to be presently undocumented. We argue that (i) allowing type variables to be implicitly introduced is confusing; and (ii) for expressiveness, both universal and existential quantifiers should be made available to programmers. Surprisingly, these language design and type inference issues seem to have received little attention in the literature, although they have most likely been "folklore" for a long time. Peyton Jones and Shields (2003) study these issues in the context of Haskell, and concur with (i). Concerning (ii), they seem to think that the language designer must choose between existential and universal type variable introduction forms—which they refer to as "type-sharing" and "type-lambda"—whereas we point out that they may and should coexist.

Polymorphic recursion

Example 1.2.10 explains how the `letrec` construct found in ML-the-programming-language may be viewed as an application of the constant `fix`, wrapped inside a normal `let` construct. Exercise 1.9.6 shows that this gives rise to a somewhat restrictive constraint generation rule: generalization occurs only *after* the application of `fix` is typechecked. In other words, in $\mathtt{letrec}\ \mathtt{f} = \lambda\mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$, all occurrences of `f` within $\mathtt{t}_1$ must have the same (monomorphic) *type*. This restriction is sometimes a nuisance, and seems unwarranted: if the function that is being defined is polymorphic, it should be possible to use it at different types even inside its own definition. Indeed, Mycroft (1984) extended Damas and Milner's type system with a more liberal treatment of recursion, commonly known as *polymorphic recursion*. The idea is to only request occurrences of $\mathtt{f}$ within $\mathtt{t}_1$ to have the same *type scheme*. Hence, they may have different *types*, all of which are instances of a common type scheme. It was later shown that well-typedness in Mycroft's extended type system is undecidable (Henglein, 1993; Kfoury, Tiuryn, and Urzyczyn, 1993). To work around this stumbling block, one solution is to use a semi-algorithm, falling back to monomorphic recursion if it does not succeed or fail in reasonable time. Although such a solution might be appealing in the setting of an automated program analysis, it is less so in the setting of a programmer-visible type system, because it may become difficult to understand why a program is ill-typed. Thus, we describe a simpler solution, which consists in requiring the programmer to explicitly supply a type scheme for $\mathtt{f}$. This is an instance of a *mandatory* type annotation.

To begin, we must change the status of $\mathtt{fix}$, because if $\mathtt{fix}$ remains a constant, then $\mathtt{f}$ must remain λ-bound and cannot receive a polymorphic type scheme. We turn $\mathtt{fix}$ into a language construct, which binds a program variable $\mathtt{f}$, and annotates it with a DM type scheme. The syntax of values and expressions is thus extended as follows:

$$\mathtt{v} ::= \ldots \mid \mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t} \qquad \mathtt{t} ::= \ldots \mid \mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t}$$

Please note that $\mathtt{f}$ is bound within $\lambda\mathtt{z}.\mathtt{t}$. The operational semantics is extended as follows.

$$(\mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t})\ \mathtt{v} \longrightarrow (\mathtt{let}\ \mathtt{f} = \mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t}\ \mathtt{in}\ \lambda\mathtt{z}.\mathtt{t})\ \mathtt{v} \qquad \text{(R-FIX')}$$

The type annotation $\mathtt{S}$ plays no essential role in the reduction; it is merely preserved. It is now possible to define $\mathtt{letrec}\ \mathtt{f} : \mathtt{S} = \lambda\mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ as syntactic sugar for $\mathtt{let}\ \mathtt{f} = \mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$.

We now give a constraint generation rule for $\mathtt{fix}$:

$$[\![\mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda\mathtt{z}.\mathtt{t} : \mathtt{T}]\!] \quad = \quad \mathsf{let}\ \mathtt{f} : \mathtt{S}\ \mathsf{in}\ [\![\lambda\mathtt{z}.\mathtt{t} : \mathtt{S}]\!] \wedge \mathtt{S} \preceq \mathtt{T}$$

The left-hand conjunct requires the function $\lambda\mathtt{z}.\mathtt{t}$ to have type scheme $\mathtt{S}$, under the assumption that $\mathtt{f}$ has type $\mathtt{S}$. Thus, it is now possible for different occurrences of $\mathtt{f}$ within $\mathtt{t}$ to receive different types. If $\mathtt{S}$ is $\forall\bar{\mathtt{X}}.\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\mathtt{t})$, then we write $[\![\mathtt{t} : \mathtt{S}]\!]$ for $\forall\bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!]$. Indeed, checking the validity of a polymorphic type annotation—be it mandatory, as is the case here, or optional, as was previously the case—requires a universally quantified constraint. The right-hand conjunct merely constrains $\mathtt{T}$ to be an instance of $\mathtt{S}$.

Given the definition of `letrec` $\mathtt{f} : \mathtt{S} = \lambda\mathtt{z}.\mathtt{t}_1$ `in` $\mathtt{t}_2$ as syntactic sugar, the above rule leads to the following derived constraint generation rule for `letrec`:

$$[\![\mathtt{letrec}\,\mathtt{f} : \mathtt{S} = \lambda\mathtt{z}.\mathtt{t}_1 \ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] \quad = \quad \mathsf{let}\ \mathtt{f} : \mathtt{S}\ \mathsf{in}\ ([\![\lambda\mathtt{z}.\mathtt{t}_1 : \mathtt{S}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{T}]\!])$$

This rule is arguably quite natural. The program variable `f` is assigned the type scheme `S` throughout its scope, that is, both inside and outside of the function's definition. The function $\lambda\mathtt{z}.\mathtt{t}_1$ must itself have type scheme `S`. Last, $\mathtt{t}_2$ must have type `T`, as in every `let` construct.

1.10.10 Exercise [★★]: Prove that the derived constraint generation rule above is indeed valid. □

It is straightforward to prove that the extended language still enjoys subject reduction. The proof relies on the following lemma: if `t` has type scheme `S`, then every instance of `S` is also a valid type for `t`.

1.10.11 Lemma: $[\![\mathtt{t} : \mathtt{S}]\!] \wedge \mathtt{S} \preceq \mathtt{T} \Vdash [\![\mathtt{t} : \mathtt{T}]\!]$. □

1.10.12 Theorem [Subject reduction]: (R-Fix') $\subseteq$ ($\sqsubseteq$). □

The programming language Haskell (Hudak, Peyton Jones, Wadler, Boutel, Fairbairn, Fasel, Guzman, Hammond, Hughes, Johnsson, Kieburtz, Nikhil, Partain, and Peterson, 1992) offers polymorphic recursion. Interesting details about its typing rules may be found in (Jones, 1999).

It is worth pointing out that some restricted instances of type inference in the presence of polymorphic recursion are decidable. This is typically the case in certain program analyses, where a type derivation for the program is already available, and the goal is only to infer extra atomic annotations, such as binding time or strictness properties. Several papers that exploit this idea are (Dussart, Henglein, and Mossin, 1995a; Jensen, 1998; Rehof and Fähndrich, 2001).

Universal types

ML-the-type-system enforces a strict stratification between types and type schemes, or, in other words, allows only prenex universal quantifiers inside types. We have pointed out earlier that there is good reason to do so: type inference for ML-the-type-system is decidable, while type inference for System F, which has no such restriction, is undecidable. Yet, this restriction comes at a cost in expressiveness: it prevents higher-order functions from accepting polymorphic function arguments, and forbids storing polymorphic functions inside data structures. Fortunately, it is in fact possible to circumvent the problem by requiring the programmer to supply additional type information.

The approach that we are about to describe is reminiscent of the way algebraic data type definitions allow circumventing the problems associated with equirecursive types (Section 1.9). Because we do not wish to extend the syntax of types with universal types of the form $\forall\bar{\mathtt{Y}}.\mathtt{T}$, we instead allow *universal type definitions*, of the form

$$\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$$

where $\mathtt{D}$ still ranges over data types. If $\mathtt{D}$ has signature $\vec{\kappa} \Rightarrow \star$, then the type variables $\vec{\mathtt{X}}$ must have kind $\vec{\kappa}$. The type $\mathtt{T}$ must have kind $\star$. The type variables $\bar{\mathtt{X}}$ and $\bar{\mathtt{Y}}$ are considered bound within $\mathtt{T}$, and the definition must be closed, that is, $ftv(\mathtt{T}) \subseteq \bar{\mathtt{X}}\bar{\mathtt{Y}}$ must hold. Last, the variance of the type constructor $\mathtt{D}$ must match its definition—a requirement stated as follows:

1.10.13 DEFINITION: Let $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \forall\bar{\mathtt{Y}}'.\mathtt{T}'$ be two α-equivalent instances of a single universal type definition, such that $\bar{\mathtt{Y}} \# ftv(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# ftv(\mathtt{T})$. Then, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \Vdash \forall\bar{\mathtt{Y}}'.\exists\bar{\mathtt{Y}}.\mathtt{T} \leq \mathtt{T}'$ must hold. □

This requirement is analogous to that found in Definition 1.9.8. The idea is, if $\mathtt{D}\,\vec{\mathtt{X}}$ and $\mathtt{D}\,\vec{\mathtt{X}}'$ are comparable, then their unfoldings $\forall\bar{\mathtt{Y}}.\mathtt{T}$ and $\forall\bar{\mathtt{Y}}'.\mathtt{T}'$ should be comparable as well. The comparison between them is expressed by the constraint $\forall\bar{\mathtt{Y}}'.\exists\bar{\mathtt{Y}}.\mathtt{T} \leq \mathtt{T}'$, which may be read: *every instance of* $\forall\bar{\mathtt{Y}}'.\mathtt{T}'$ *is (a supertype of) an instance of* $\forall\bar{\mathtt{Y}}.\mathtt{T}$. Again, when subtyping is interpreted as equality, the requirement of Definition 1.10.13 is always satisfied; it becomes nontrivial only in the presence of true subtyping.

The effect of the universal type definition $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ is to enrich the programming language with a new construct:

$$\mathtt{v} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\ \mathtt{v} \qquad \mathtt{t} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\ \mathtt{t} \qquad \mathcal{E} ::= \ldots \mid \mathtt{pack}_{\mathtt{D}}\ \mathcal{E}$$

and with a new unary destructor $\mathtt{open}_{\mathtt{D}}$. Their operational semantics is as follows:

$$\mathtt{open}_{\mathtt{D}}\ (\mathtt{pack}_{\mathtt{D}}\ \mathtt{v}) \xrightarrow{\delta} \mathtt{v} \qquad \text{(R-OPEN-ALL)}$$

Intuitively, $\mathtt{pack}_{\mathtt{D}}$ and $\mathtt{open}_{\mathtt{D}}$ are the two coercions that witness the isomorphism between $\mathtt{D}\,\vec{\mathtt{X}}$ and $\forall\bar{\mathtt{Y}}.\mathtt{T}$. The value $\mathtt{pack}_{\mathtt{D}}\ \mathtt{v}$ behaves exactly like $\mathtt{v}$, except it is marked, as a hint to the typechecker. As a result, the mark must be removed using $\mathtt{open}_{\mathtt{D}}$ before the value can be used.

What are the typing rules for $\mathtt{pack}_{\mathtt{D}}$ and $\mathtt{open}_{\mathtt{D}}$? In System F, they would receive types $\forall\bar{\mathtt{X}}.(\forall\bar{\mathtt{Y}}.\mathtt{T}) \rightarrow \mathtt{D}\,\vec{\mathtt{X}}$ and $\forall\bar{\mathtt{X}}.\mathtt{D}\,\vec{\mathtt{X}} \rightarrow \forall\bar{\mathtt{Y}}.\mathtt{T}$, respectively. However, neither of these is a valid type scheme: both exhibit a universal quantifier under an arrow.

In the case of $\mathtt{pack}_{\mathtt{D}}$, which has been made a language construct rather than a constant, we work around the problem by embedding this universal quantifier in the constraint generation rule:

$$[\![\mathtt{pack}_\mathtt{D}\ \mathtt{t} : \mathtt{T}']\!] \quad = \quad \exists\bar{\mathtt{X}}.([\![\mathtt{t} : \forall\bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{T}')$$

The rule implicitly requires that $\bar{\mathtt{X}}$ be fresh for the left-hand side and that $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ be (an α-variant of) the definition of $\mathtt{D}$. The left-hand conjunct requires $\mathtt{t}$ to have type scheme $\forall\bar{\mathtt{Y}}.\mathtt{T}$. The notation $[\![\mathtt{t} : \mathtt{S}]\!]$ was defined on page 114. The right-hand conjunct states that a valid type for $\mathtt{pack}_\mathtt{D}\ \mathtt{t}$ is (a supertype of) $\mathtt{D}\,\vec{\mathtt{X}}$.

We deal with $\mathtt{open}_\mathtt{D}$ as follows. Provided $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$, we extend the initial environment Γ_0 with the binding $\mathtt{open}_\mathtt{D} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}.\mathtt{D}\,\vec{\mathtt{X}} \to \mathtt{T}$. We have simply hoisted the universal quantifier outside of the arrow—a valid isomorphism in System F.

The proof of the subject reduction theorem must be extended with the following new case:

1.10.14 Theorem [Subject reduction]: (R-Open-All) $\subseteq$ ($\sqsubseteq$). □

Proof: We have

$$\begin{array}{lll}
 & \text{let } \Gamma_0 \text{ in } [\![\mathtt{open}_\mathtt{D}\ (\mathtt{pack}_\mathtt{D}\ \mathtt{v}) : \mathtt{T}_0]\!] & \\
\equiv & \text{let } \Gamma_0 \text{ in } \exists\mathtt{Z}.(\mathtt{open}_\mathtt{D} \preceq \mathtt{Z} \to \mathtt{T}_0 \wedge [\![\mathtt{pack}_\mathtt{D}\ \mathtt{v} : \mathtt{Z}]\!]) & \mathbf{(1)}\\
\equiv & \text{let } \Gamma_0 \text{ in } \exists\mathtt{Z}.(\exists\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.(\mathtt{D}\,\vec{\mathtt{X}}' \to \mathtt{T}' \leq \mathtt{Z} \to \mathtt{T}_0) \wedge \exists\bar{\mathtt{X}}.([\![\mathtt{v} : \forall\bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{Z})) & \mathbf{(2)}\\
\equiv & \text{let } \Gamma_0 \text{ in } \exists\bar{\mathtt{X}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.([\![\mathtt{v} : \forall\bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \wedge \mathtt{T}' \leq \mathtt{T}_0) & \mathbf{(3)}\\
\Vdash & \text{let } \Gamma_0 \text{ in } \exists\bar{\mathtt{X}}\bar{\mathtt{Y}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.([\![\mathtt{v} : \forall\bar{\mathtt{Y}}.\mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}' \wedge \mathtt{T}' \leq \mathtt{T}_0) & \mathbf{(4)}\\
\Vdash & \text{let } \Gamma_0 \text{ in } \exists\bar{\mathtt{X}}\bar{\mathtt{Y}}\bar{\mathtt{X}}'\bar{\mathtt{Y}}'.[\![\mathtt{v} : \mathtt{T}_0]\!] & \mathbf{(5)}\\
\equiv & \text{let } \Gamma_0 \text{ in } [\![\mathtt{v} : \mathtt{T}_0]\!] & \mathbf{(6)}
\end{array}$$

where (1) is by definition of constraint generation for applications and for constants; $\mathtt{Z}$ is fresh; (2) is by definition of constraint generation for $\mathtt{pack}_\mathtt{D}$ and $\mathtt{open}_\mathtt{D}$, where $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \forall\bar{\mathtt{Y}}'.\mathtt{T}'$ are two α-equivalent instances of the definition of $\mathtt{D}$; $\bar{\mathtt{X}}$, $\bar{\mathtt{Y}}$, $\bar{\mathtt{X}}'$, and $\bar{\mathtt{Y}}'$ are fresh and satisfy $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# \mathit{ftv}(\mathtt{T})$; (3) is by C-ExAnd, C-Arrow, and C-ExTrans, which allows eliminating $\mathtt{Z}$; (4) is by Definition 1.10.13, Lemma 1.10.1, and C-ExAnd; (5) is by Lemmas 1.10.11 and 1.6.3; (6) is by C-Ex*. □

The proof of (R-Context) $\subseteq$ ($\sqsubseteq$) must also be extended with a new subcase, corresponding the new production $\mathcal{E} ::= \ldots \mid \mathtt{pack}_\mathtt{D}\ \mathcal{E}$. If the language is pure, this is straightforward. In the presence of side effects, however, this subcase fails, because universal and existential quantifiers in constraints do not commute. The problem is then avoided by restricting $\mathtt{pack}_\mathtt{D}$ to values, as in Definition 1.7.7.

This approach to extending ML-the-type-system with universal (or existential—see below) types has been studied in (Läufer and Odersky, 1994;

Rémy, 1994; Odersky and Läufer, 1996; Shields and Peyton Jones, 2002). Laüfer and Odersky have suggested combining universal or existential type declarations with algebraic data type definitions. This allows suppressing the cumbersome $\mathtt{pack}_{\mathtt{D}}$ and $\mathtt{open}_{\mathtt{D}}$ constructs; instead, one simply uses the standard syntax for constructing and deconstructing variants and records.

Existential types

Existential types (TAPL Chapter 24) are close cousins of universal types, and may be introduced into ML-the-type-system in the same manner. Actually, existential types have been introduced in ML-the-type-system before universal types. We give a brief description of this extension, insisting mainly on the differences with the case of universal types.

We now allow *existential type definitions*, of the form $\mathtt{D}\,\vec{\mathtt{X}} \approx \exists\bar{\mathtt{Y}}.\mathtt{T}$. The conditions required of a well-formed definition are unchanged, except the variance requirement, which is dual:

1.10.15 Definition: Let $\mathtt{D}\,\vec{\mathtt{X}} \approx \exists\bar{\mathtt{Y}}.\mathtt{T}$ and $\mathtt{D}\,\vec{\mathtt{X}}' \approx \exists\bar{\mathtt{Y}}'.\mathtt{T}'$ be two α-equivalent instances of a single existential type definition, such that $\bar{\mathtt{Y}} \# \mathit{ftv}(\mathtt{T}')$ and $\bar{\mathtt{Y}}' \# \mathit{ftv}(\mathtt{T})$. Then, $\mathtt{D}\,\vec{\mathtt{X}} \leq \mathtt{D}\,\vec{\mathtt{X}}' \Vdash \forall\bar{\mathtt{Y}}.\exists\bar{\mathtt{Y}}'.\mathtt{T} \leq \mathtt{T}'$ must hold. □

The effect of this existential type definition is to enrich the programming language with a new unary constructor $\mathtt{pack}_{\mathtt{D}}$ and with a new construct: $\mathtt{t} ::= \ldots \mid \mathtt{open}_{\mathtt{D}}\ \mathtt{t}\ \mathtt{t}$ and $\mathcal{E} ::= \ldots \mid \mathtt{open}_{\mathtt{D}}\ \mathcal{E}\ \mathtt{t} \mid \mathtt{open}_{\mathtt{D}}\ \mathtt{v}\ \mathcal{E}$. Their operational semantics is as follows:

$$\mathtt{open}_{\mathtt{D}}\ (\mathtt{pack}_{\mathtt{D}}\mathtt{v}_1)\ \mathtt{v}_2 \longrightarrow \mathtt{v}_2\ \mathtt{v}_1 \qquad \text{(R-Open-Ex)}$$

In the literature, the second argument of $\mathtt{open}_{\mathtt{D}}$ is often required to be a λ-abstraction $\lambda\mathtt{z.t}$, so the construct becomes $\mathtt{open}_{\mathtt{D}}\ \mathtt{t}\ (\lambda\mathtt{z.t})$, often written $\mathtt{open}_{\mathtt{D}}\ \mathtt{t}\ \mathtt{as}\ \mathtt{z}\ \mathtt{in}\ \mathtt{t}$.

Provided $\bar{\mathtt{X}} \# \bar{\mathtt{Y}}$, we extend the initial environment Γ_0 with the binding $\mathtt{pack}_{\mathtt{D}} : \forall\bar{\mathtt{X}}\bar{\mathtt{Y}}.\mathtt{T} \rightarrow \mathtt{D}\,\vec{\mathtt{X}}$. The constraint generation rule for $\mathtt{open}_{\mathtt{D}}$ is as follows:

$$[\![\mathtt{open}_{\mathtt{D}}\ \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}']\!] \quad = \quad \exists\bar{\mathtt{X}}.([\![\mathtt{t}_1 : \mathtt{D}\,\vec{\mathtt{X}}]\!] \wedge [\![\mathtt{t}_2 : \forall\bar{\mathtt{Y}}.\mathtt{T} \rightarrow \mathtt{T}']\!])$$

The rule implicitly requires that $\bar{\mathtt{X}}$ be fresh for the left-hand side, that $\bar{\mathtt{Y}}$ be fresh for $\mathtt{T}'$, and that $\mathtt{D}\,\vec{\mathtt{X}} \approx \forall\bar{\mathtt{Y}}.\mathtt{T}$ be (an α-variant of) the definition of $\mathtt{D}$. The left-hand conjunct simply requires $\mathtt{t}_1$ to have type $\mathtt{D}\,\vec{\mathtt{X}}$. The right-hand conjunct states that the function $\mathtt{t}_2$ must be prepared to accept an argument of type $\mathtt{T}$, for *any* $\bar{\mathtt{Y}}$, and produce a result of the expected type $\mathtt{T}'$. In other words, $\mathtt{t}_2$ must be a polymorphic function.

The type scheme of existential $\mathtt{pack}_{\mathtt{D}}$ resembles that of universal $\mathtt{open}_{\mathtt{D}}$, while the constraint generation rule for existential $\mathtt{open}_{\mathtt{D}}$ is a close cousin of that for universal $\mathtt{pack}_\mathtt{D}$. Thus, the duality between universal and existential types is rather strong. The main difference lies in the fact that the existential $\mathtt{open}_\mathtt{D}$ construct is *binary*, rather than unary, so as to limit the scope of the newly introduced type variables $\bar{\mathtt{Y}}$. The duality may be better understood by studying the encoding of existential types in terms of universal types (Reynolds, 1983b).

As expected, R-Open-Ex preserves types.

1.10.16 Theorem [Subject reduction]: (R-Open-Ex) $\subseteq$ ($\sqsubseteq$). □

1.10.17 Exercise [★★, $\not\twoheadrightarrow$]: Prove Theorem 1.10.16. The proof is analogous, although not identical, to that of Theorem 1.10.14. □

In the presence of side effects, the new production $\mathcal{E} ::= \ldots \mid \mathtt{open}_\mathtt{D}\ \mathtt{v}\ \mathcal{E}$ is problematic. The standard workaround is to restrict the second argument to $\mathtt{open}_\mathtt{D}$ to be a value.

1.11 Rows

In Section 1.9, we have shown how to extend ML-the-programming-language with algebraic data types, that is, variant and record type definitions, which we now refer to as *simple*. This mechanism has a severe limitation: two distinct definitions must define incompatible types. As a result, one cannot hope to write code that uniformly operates over variants or records of different shapes, because the type of such code is not even expressible.

For instance, it is impossible to express the type of the *polymorphic record access* operation, which retrieves the value stored at a particular field ℓ inside a record, *regardless* of which other fields are present. Indeed, if the label ℓ appears with type $\mathtt{T}$ in the definition of the simple record type $\mathtt{D}\,\vec{\mathtt{X}}$, then the associated record access operation has type $\forall\bar{\mathtt{X}}.\mathtt{D}\,\vec{\mathtt{X}} \rightarrow \mathtt{T}$. If ℓ appears with type $\mathtt{T}'$ in the definition of another simple record type, say $\mathtt{D}'\,\vec{\mathtt{X}}'$, then the associated record access operation has type $\forall\bar{\mathtt{X}}'.\mathtt{D}'\,\vec{\mathtt{X}}' \rightarrow \mathtt{T}'$; and so on. The most precise type scheme that subsumes all of these incomparable type schemes is $\forall\mathtt{XY}.\mathtt{X} \rightarrow \mathtt{Y}$. It is, however, not a sound type scheme for the record access operation. Another powerful operation whose type is currently not expressible is *polymorphic record extension*, which copies a record and stores a value at field ℓ in the copy, possibly creating the field if it did not previously exist, again *regardless* of which other fields are present. (If ℓ was known to previously exist, the operation is known as *polymorphic record update*.)

In order to assign types to polymorphic record operations, we must do away with record type definitions: we must replace *named* record types, such as $\mathtt{D}\,\vec{\mathtt{X}}$, with *structural* record types that provide a direct description of the record's domain and contents. (Following the analogy between a record and a partial function from labels to values, we use the word *domain* to refer to the set of fields that are defined in a record.) For instance, a product type is structural: the type $\mathtt{T}_1 \times \mathtt{T}_2$ is the (undeclared) type of pairs whose first component has type $\mathtt{T}_1$ and whose second component has type $\mathtt{T}_2$. Thus, we wish to design record types that behave very much like product types. In doing so, we face two orthogonal difficulties. First, as opposed to pairs, records may have different domains. Because the type system must statically ensure that no undefined field is accessed, information about a record's domain must be made part of its type. Second, because we suppress record type definitions, labels must now be predefined. However, for efficiency and modularity reasons, it is impossible to explicitly list *every* label in existence in every record type.

In what follows, we explain how to address the first difficulty in the simple setting of a finite set of labels. Then, we introduce *rows*, which allow dealing with an infinite set of labels, and address the second difficulty. We define the syntax and logical interpretation of rows, study the new constraint equivalence laws that arise in their presence, and extend the first-order unification algorithm with support for rows. Then, we review several applications of rows, including polymorphic operations on records, variants, and objects, and discuss alternatives to rows.

Records with finite carrier

Let us temporarily assume that $\mathcal{L}$ is finite. In fact, for the sake of definiteness, let us assume that $\mathcal{L}$ is $\{\ell_a, \ell_b, \ell_c\}$.

To begin, let us consider only *full* records, whose domain is exactly $\mathcal{L}$—in other words, tuples indexed by $\mathcal{L}$. To describe them, it is natural to introduce a type constructor record of signature $\star \otimes \star \otimes \star \Rightarrow \star$. The type $\mathsf{record}\ \mathtt{T}_a\ \mathtt{T}_b\ \mathtt{T}_c$ represents all records where the field ℓ_a (resp. ℓ_b, ℓ_c) contains a value of type $\mathtt{T}_a$ (resp. $\mathtt{T}_b$, $\mathtt{T}_c$). Please note that record is nothing but a product type constructor of arity 3. The basic operations on records, namely *creation* of a record out of a default value, which is stored into every field, *update* of a particular field (say, ℓ_b), and *access* to a particular field (say, ℓ_b), may be assigned the following type schemes:

$$\begin{array}{rl} \{\cdot\} : & \forall \mathtt{X}.\mathtt{X} \rightarrow \mathsf{record}\ \mathtt{X}\ \mathtt{X}\ \mathtt{X} \\ \{\cdot \text{ with } \ell_b = \cdot\} : & \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}'_b\mathtt{X}_c.\mathsf{record}\ \mathtt{X}_a\ \mathtt{X}_b\ \mathtt{X}_c \rightarrow \mathtt{X}'_b \rightarrow \mathsf{record}\ \mathtt{X}_a\ \mathtt{X}'_b\ \mathtt{X}_c \\ \cdot.\{\ell_b\} : & \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}_c.\mathsf{record}\ \mathtt{X}_a\ \mathtt{X}_b\ \mathtt{X}_c \rightarrow \mathtt{X}_b \end{array}$$

Here, polymorphism allows updating or accessing a field without knowledge of the types of the other fields. This flexibility is made possible by the property that all record types are formed using a single record type constructor.

This is fine, but in general, the domain of a record is not necessarily $\mathcal{L}$: it may be a subset of $\mathcal{L}$. How may we deal with this fact, while maintaining the above key property? A naïve approach consists in encoding arbitrary records in terms of full records, using the standard algebraic data type option, whose definition is $\mathsf{option}\,\mathtt{X} \approx \mathsf{pre}\ \mathtt{X} + \mathsf{abs}$. We use pre for *present* and abs for *absent*: indeed, a field that is defined with value $\mathtt{v}$ is encoded as a field with value $\mathsf{pre}\ \mathtt{v}$, while an undefined field is encoded as a field with value abs. Thus, an arbitrary record whose fields, *if present*, have types $\mathtt{T}_a$, $\mathtt{T}_b$, and $\mathtt{T}_c$, respectively, may be encoded as a full record of type $\mathsf{record}\ (\mathsf{option}\ \mathtt{T}_a)\ (\mathsf{option}\ \mathtt{T}_b)\ (\mathsf{option}\ \mathtt{T}_c)$. This naive approach suffers from a serious drawback: record types still contain no domain information. As a result, field access must involve a dynamic check, so as to determine whether the desired field is present: in our encoding, this corresponds to the use of $\mathtt{case}_{\mathsf{option}}$.

To avoid this overhead and increase programming safety, we must move this check from runtime to compile time. In other words, we must make the type system aware of the difference between pre and abs. To do so, we replace the definition of option by two separate algebraic data type definitions, namely $\mathsf{pre}\,\mathtt{X} \approx \mathsf{pre}\ \mathtt{X}$ and $\mathsf{abs} \approx \mathsf{abs}$. In other words, we introduce a unary type constructor pre, whose only associated data constructor is pre, and a nullary type constructor abs, whose only associated data constructor is abs. Record types now contain domain information: for instance, a record of type $\mathsf{record}\ \mathsf{abs}\ (\mathsf{pre}\ \mathtt{T}_b)\ (\mathsf{pre}\ \mathtt{T}_c)$ must have domain $\{\ell_b, \ell_c\}$. Thus, the type of a field tells whether it is defined. Since the type pre has no data constructors other than pre, the accessor pre^{-1}, whose type is $\forall \mathtt{X}.\mathsf{pre}\ \mathtt{X} \rightarrow \mathtt{X}$, and which allows retrieving the value stored in a field, cannot fail. Thus, the dynamic check has been eliminated.

To complete the definition of our encoding, we now define operations on arbitrary records in terms of operations on full records. To distinguish between the two, we write the former with angle braces, instead of curly braces. The *empty record* $\langle\rangle$, where all fields are undefined, may be defined as $\{\mathsf{abs}\}$. *Extension* at a particular field (say, ℓ_b) $\langle\cdot\ \mathsf{with}\ \ell_b = \cdot\rangle$ is defined as $\lambda\mathtt{r}.\lambda\mathtt{z}.\{\mathtt{r}\ \mathsf{with}\ \ell_b = \mathsf{pre}\ \mathtt{z}\}$. *Access* at a particular field (say, ℓ_b) $\cdot.\langle\ell_b\rangle$ is defined as $\lambda\mathtt{z}.\mathsf{pre}^{-1}\mathtt{z}.\{\ell_b\}$. It is straightforward to check that these operations have the following principal type schemes:

$$\begin{aligned}
\langle\rangle :&\quad \mathsf{record}\ \mathsf{abs}\ \mathsf{abs}\ \mathsf{abs}\\
\langle\cdot\ \mathsf{with}\ \ell_b = \cdot\rangle :&\quad \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}'_b\mathtt{X}_c.\mathsf{record}\ \mathtt{X}_a\ \mathtt{X}_b\ \mathtt{X}_c \rightarrow \mathtt{X}'_b \rightarrow \mathsf{record}\ \mathtt{X}_a\ (\mathsf{pre}\ \ \mathtt{X}'_b)\ \mathtt{X}_c\\
\cdot.\langle\ell_b\rangle :&\quad \forall \mathtt{X}_a\mathtt{X}_b\mathtt{X}_c.\mathsf{record}\ \mathtt{X}_a\ (\mathsf{pre}\ \ \mathtt{X}_b)\ \mathtt{X}_c \rightarrow \mathtt{X}_b
\end{aligned}$$

It is important to notice that the type schemes associated with extension and access at ℓ_b are polymorphic in $\mathtt{X}_a$ and $\mathtt{X}_c$, which now means that *these operations are insensitive not only to the type, but also to the presence or* *absence of the fields ℓ_a and ℓ_c*. Furthermore, extension is polymorphic in $\mathtt{X}_b$, which means that it is insensitive to the presence or absence of the field ℓ_b in its argument. The subterm $\mathsf{pre}\,\mathtt{X}'_b$ in its result type reflects the fact that ℓ_b is defined in the extended record. Conversely, the subterm $\mathsf{pre}\,\mathtt{X}_b$ in the type of the access operation reflects the requirement that ℓ_b be defined in its argument.

Our encoding of arbitrary records in terms of full records was carried out for pedagogical purposes. In practice, no such encoding is necessary: the *data* constructors pre and abs have no machine representation, and the compiler is free to lay out records in memory in an efficient manner. The encoding is interesting, however, because it provides a natural way of introducing the *type* constructors pre and abs, which play an important role in our treatment of polymorphic record operations.

We remark that, in our encoding, the arguments of the type constructor record are expected to be either type variables or formed with pre or abs, while, conversely, the type constructors pre and abs are not intended to appear anywhere else. It is possible to enforce this invariant using kinds. In addition to $\star$, let us introduce the kind $\diamond$ of *field types*. Then, let us adopt the following signatures: $\mathsf{pre}: \star \Rightarrow \diamond$, $\mathsf{abs} : \diamond$, and $\mathsf{record} : \diamond \otimes \diamond \otimes \diamond \Rightarrow \star$.

1.11.1 Exercise [★, Recommended, $\nrightarrow$]: Check that the three type schemes given above are well-kinded. What is the kind of each type variable? □

1.11.2 Exercise [★★, Recommended, $\nrightarrow$]: Our record types contain information about every field, regardless of whether it is defined: we encode definedness information within the type of each field, using the type constructors pre and abs. A perhaps more natural approach would be to introduce a family of record type constructors, indexed by the subsets of $\mathcal{L}$, so that the types of records with different domains are formed with different constructors. For instance, the empty record would have type $\{\}$; a record that defines the field ℓ_a only would have a type of the form $\{\ell_a : \mathtt{T}_a\}$; a record that defines the fields ℓ_b and ℓ_c only would have a type of the form $\{\ell_b : \mathtt{T}_b; \ell_c : \mathtt{T}_c\}$; and so on. Assuming that the type discipline is Damas and Milner's (that is, assuming an equality-only syntactic model), would it be possible to assign satisfactory type schemes to polymorphic record access and extension? Would it help to equip record types with a nontrivial subtyping relation? □

Records with infinite carrier

Finite records are insufficient both from practical and theoretical points of view. In practice, the set of labels could become very large, making the type of every record as large as the set of labels itself, even if only a few labels are actually defined. In principle, the set of labels could even be infinite. Actually, in modular programs the whole set of labels may not be known in advance, which amounts in some way to working with an infinite set of labels. Thus, records must be drawn from an infinite set of labels—whether their domains are finite or infinite. Still, we can restrict our attention to records that are almost constant, that is, records where only a finite number of fields differ. With this restriction, full records (defined everywhere) can always be built by giving explicit definitions for a finite number of fields and a default value for all other fields, as in the finite case. For instance, the record $\{\{\{\texttt{false}\}\ \textsf{with}\ \ell = 1\}\ \textsf{with}\ \ell' = \texttt{true}\}$ is the record equal to `true` on field ℓ', to 1 on field ℓ, and to `false` on any other field.

Types of records are functions from labels to types, called *rows*. However, for sake of generality, we use a unary type constructor, say Π, as an indirection between rows and record types. Moreover, we further restrict our attention to the case where rows are also almost constant. (The fact that the property holds for record values does not imply that it also holds for record types, for the default value of some record could have a polymorphic type, and one could wish to see each field with a different instance of this polymorphic type. So this is a true restriction, but a reasonable one.) Thus, rows can also be represented by giving explicit types for a finite number of fields and a default type for all other fields. We write $\partial\mathtt{T}$ the row whose type is $\mathtt{T}$ on every field, and $(\ell : \mathtt{T}\ ;\ \mathtt{T}')$ the row whose type is $\mathtt{T}$ on field ℓ and $\mathtt{T}'$ on other fields. Formally, ∂ is a unary type constructor and ℓ is a family of binary type constructors, written with syntactic sugar $(\ell : \cdot\ ;\ \cdot)$. For example, $\Pi(\ell : \mathtt{bool}\ ;\ (\ell' : \mathtt{int}\ ;\ \partial\mathtt{bool}))$ is a record type that describes records whose field ℓ carries a value of type `bool`, field ℓ' carries a value of type `int`, and all other fields carry values of type `bool`. In fact, this is a sound type for the record defined above. In fact, the type $\Pi(\ell' : \mathtt{int}\ ;\ (\ell : \mathtt{bool}\ ;\ \partial\mathtt{bool}))$ should also be a sound type for this record, since the order in which fields are specified should not matter. We actually treat both types as equivalent. Furthermore, the row $(\ell : \mathtt{bool}\ ;\ \partial\mathtt{bool})$, which stands for `bool` on field ℓ and $\partial\mathtt{bool}$ everywhere else, must also be equivalent to $\partial\mathtt{bool}$, which stands for `bool` everywhere.

A record type may also contain type variables. For instance, the record $\lambda\mathtt{z}.\{\mathtt{z}\}$ that maps any value $\mathtt{v}$ to a record with the default value $\mathtt{v}$ has type $\mathtt{X} \rightarrow \Pi(\partial\mathtt{X})$. Projections of this record on any field will return a value of the same type $\mathtt{X}$. By comparison, the function that reads some field ℓ of its (record) argument has type $\Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \rightarrow \mathtt{X}$: this says that the argument must be a record where field ℓ has type $\mathtt{X}$ and other fields may have *any* type. Variable $\mathtt{Y}$ is called a *row variable*, since it can be instantiated to any row. For instance, $\mathtt{Y}$ can be instantiated to $(\ell' : \mathtt{int}\ ;\ \mathtt{Y}')$ and as a result this function can be applied to the record above. Conversely, the row $\partial\mathtt{X}$, which is equal to $(\ell' : \mathtt{X}\ ;\ \mathtt{X})$, can only be instantiated to rows of the form $\partial\mathtt{T}$, which are equal to $(\ell' : \mathtt{T} \mathrel{;} \mathtt{T})$, that is, to constant rows.

Syntax of row types

Let $\mathcal{L}$ be a denumerable collection of labels. We write $\ell.L$ for $\{\ell\} \uplus L$, which implies $\ell \notin L$. We first introduce kinds, so as to distinguish rows such as $(\ell : \mathtt{int} \mathrel{;} \partial\mathtt{bool})$ from *basic types*, such as $\mathtt{int}$ or $\mathtt{int} \to \mathtt{int}$.

1.11.3 Definition [Row kinds]: Let *row kinds* be composed of a particular kind *Type* and the collection of kinds $Row(L)$ where L ranges over finite subsets of $\mathcal{L}$. We use letter s to range over row kinds. □

Intuitively, a row of kind $Row(L)$ is a function of domain $\mathcal{L} \setminus L$ to types. That is, L specifies the set of labels that the row *must not* define. For instance, the (basic) type $\Pi(\ell : \mathtt{int} \mathrel{;} \mathtt{X})$ has kind *Type*, the row $(\ell : \mathtt{int} \mathrel{;} \mathtt{X})$ has kind $Row(\emptyset)$ provided $\mathtt{X}$ has kind $Row(\{\ell\})$.

To remain abstract the definition of rows is parameterized by a signature $\mathcal{S}_0$ for building basic types and a signature $\mathcal{S}_1$ for building rows. From those two signatures, we then define a new signature $\mathcal{S}$ that completely specifies the set of types. However, the signature $\mathcal{S}$ must superimposed row kinds on top of the (basic) kinds of the two input signatures $\mathcal{S}_0$ and $\mathcal{S}_1$. We use product signatures to enlighten this process. More precisely, we build a product signature from two signatures $K \Rightarrow \kappa$ and $K' \Rightarrow \kappa'$ with the following notations: we write $\kappa.\kappa'$ for the pair (κ, κ'); $K.\kappa$ for the mapping $(d \mapsto K(d).\kappa)^{d \in dom(K)}$; $(K \Rightarrow \kappa).\kappa'$ for the kind signature $K.\kappa \Rightarrow \kappa.\kappa'$; and symmetrically, we write $\kappa.K'$ and $\kappa.(K' \Rightarrow \kappa')$. The signature $\mathcal{S}$ reuses the same input type constructors as $\mathcal{S}_0$ and $\mathcal{S}_1$, but at different row kinds. We use superscripts to provide copies of type constructors at different kinds, and thus avoid overloading of kinds.

1.11.4 Definition [Row extension of a signature]: Let $\mathcal{S}_0$ and $\mathcal{S}_1$ be signatures where all symbols of $\mathcal{S}_1$ are unary. The row extension of $\mathcal{S}_0$ with $\mathcal{S}_1$ is the signature $\mathcal{S}$ defined as follows where κ ranges over basic kinds (those used in $\mathcal{S}_0$ and $\mathcal{S}_1$) and s ranges over row kinds:

$F \in dom(\mathcal{S})$	Signature	Conditions
G^s	$(K \Rightarrow \kappa).s$	$(G : K \Rightarrow \kappa) \in \mathcal{S}_0$
H	$K.Row(\emptyset) \Rightarrow \kappa.Type$	$(H : K \Rightarrow \kappa) \in \mathcal{S}_1$
$\partial^{\kappa,L}$	$\kappa.(Type \Rightarrow Row(L))$	
$\ell^{\kappa,L}$	$\kappa.(Type \otimes Row(\ell.L) \Rightarrow Row(L))$	$\ell \notin L$

□

We usually write $\ell^{\kappa,L} : \mathtt{t}_1 \mathrel{;} \mathtt{t}_2$ instead of $\ell^{\kappa,L}\ \mathtt{t}_1\ \mathtt{t}_2$ and let this symbol be right-associative. We often drop the superscripts of type constructors since, for any type expression $\mathtt{T}$, superscripts can be unambiguously recovered from the kind of $\mathtt{T}$.

1.11.5 Example: Let us assume there is a single basic kind $\star$ and that $\mathcal{S}_1$ contain a unique type constructor Π (hence of kind $\star \Rightarrow \star$). An example of row type is $\mathtt{X}_0 \rightarrow \Pi(\ell_1 : G \; ; (\mathtt{Y} \rightarrow \partial \mathtt{X}_0))$. With all superscripts annotations, this type is

$$\mathtt{X}_0 \rightarrow^{\star, \mathit{Type}} \Pi({\ell_1}^{\star, \mathit{Row}(\emptyset)} : G^{\mathit{Type}} \; ; (\mathtt{Y} \rightarrow^{\star, \mathit{Row}(\{\ell_1\})} \partial^{\star, \mathit{Row}(\{\ell_1\})} \mathtt{X}_0)).$$

Intuitively, this is the type of a function that takes a value of type $\mathtt{X}_0$ and returns a record where field ℓ_1 has type G and all other fields are functions that given a value of an arbitrary type would returns a value of (the same) type $\mathtt{X}_0$. An instance of this type is $\mathtt{X}_0 \rightarrow \Pi(\ell_1 : G \; ; ((\ell_2 : \mathtt{Y}_2 \; ; \mathtt{Y}') \rightarrow (\ell_2 : \mathtt{X}_0 \; ; \partial \mathtt{X}_0)))$, obtained by instantiating row variable $\mathtt{Y}$ and by expanding the constant row $\partial \mathtt{X}_0$. As shown below, this type is actually equivalent to $\mathtt{X}_0 \rightarrow \Pi(\ell_1 : G \; ; \ell_2 : \mathtt{Y}_2 \rightarrow \mathtt{X}_0 \; ; (\mathtt{Y}' \rightarrow \partial \mathtt{X}_0))$, by distributivity of type constructor $\rightarrow$ other type constructor ℓ_2. Please, note again the difference between $\mathtt{Y}$, which is a row variable that can expand to different type variables on different labels, and $\partial \mathtt{X}$, which is a constant row that expands to the same type variable $\mathtt{X}$ on all labels. □

1.11.6 Example [Ill-kinded expression]: Under the assumptions of the previous example, the expression $\mathtt{X} \rightarrow \Pi(\mathtt{X})$ is not a row type, since variable $\mathtt{X}$ cannot simultaneously be of row kind *Type* and $\mathit{Row}(\emptyset)$ as required by its two occurrences, from left to right respectively. The expression $(\ell : \mathtt{X} \; ; \ell : \mathtt{X}' \; ; \mathtt{X}'')$ is also ill-kinded, for the inner expression $(\ell : \mathtt{X}' \; ; \mathtt{X}'')$ of row kind $\mathit{Row}(L)$ with $\ell \notin L$ cannot also have row kind $\mathit{Row}(\{\ell\})$, as required by its occurrence in the whole expression. Indeed, row kinds prohibit multiple definitions of the same label, as well as using rows in place of basic types and conversely. Notice that $\Pi(\Pi(\mathtt{X}))$ is also ill-formed, since type constructors of $\mathcal{S}_1$ are not lifted to row kinds and thus cannot appear in rows, except under the type constructor ∂, hence as basic types. □

1.11.7 Exercise [★★★, $\not\rightarrow$]: Design an algorithm that infers superscripts of type constructors of a type expression from its kind. Can the kind of the expression be inferred as well? Can you give an algorithm to check that type expressions are well-kinded when both the superscripts of type constructors and the kinds of the whole type expressions are omitted? □

Meaning of rows

As mentioned above, a row of kind $\mathit{Row}(L)$ stands for a function from $\mathcal{L} \setminus L$ to types. Actually, it is simpler to represent this function explicitly as an infinitely branching tree in the model. For this purpose, we use a collection of constructors L of (infinite but denumerable) arity $\mathcal{L} \setminus L$.

1.11.8 Definition [Row model]: Let $\mathcal{S}$ be the row extension of a signature $\mathcal{S}_0$ with a signature $\mathcal{S}_1$. Let $\mathcal{S}_{\mathcal{M}}$ be the following signature, where κ ranges over basic kinds and L ranges over finite subsets of $\mathcal{L}$:

$F \in \mathit{dom}(\mathcal{S}_{\mathcal{M}})$	Signature	Conditions
G	$(K \Rightarrow \kappa).\mathit{Type}$	$(G : K \Rightarrow \kappa) \in \mathcal{S}_0$
H	$K.\mathit{Row}(\emptyset) \Rightarrow \kappa.\mathit{Type}$	$(H : K \Rightarrow \kappa) \in \mathcal{S}_1$
L^κ	$\kappa.(\mathit{Type}^{\mathcal{L} \setminus L} \Rightarrow \mathit{Row}(L))$	

Let $\mathcal{M}_\kappa$ consist of the regular trees t built over the signature $\mathcal{S}_{\mathcal{M}}$ such that $t(\epsilon)$ has image kind κ. We interpret a type constructor F of signature $K \Rightarrow \kappa.s$ as a function that maps $T \in \mathcal{M}_K$ to $t \in \mathcal{M}_{\kappa.s}$ defined by cases on F and on the basic kind κ.

$F \in \mathcal{S}$	$t(\epsilon)$	For $d \in \mathit{dom}(K)$ and $\ell \in \mathcal{L} \setminus L, \ell \neq \ell_0$.
G^{Type}	G	$t/d = T(d)$
H	H	$t/1 = T(1)$
$G^{\mathit{Row}(L)}$	L^κ	$t(\ell) = G \wedge t/(\ell \cdot d) = T(d)/\ell$
$\partial^{\kappa,L}$	L^κ	$t/\ell = T(1)$
$\ell_0^{\kappa,L}$	L^κ	$t/\ell_0 = T(1) \wedge t/\ell = T(2)/\ell$

In the presence of subtyping, we let type constructors G and H behave in $\mathcal{S}_{\mathcal{M}}$ as in $\mathcal{S}_0$ and $\mathcal{S}_1$ and type constructors L^κ be isolated and covariant in every position. Models that define ground types and interpret type constructors in this manner are referred to as *row* models. □

Reasoning with row types

In this section, we assume a subtyping model. All reasoning principles also apply to the equality-only model, which is a subcase of the subtyping model.

The meaning of rows has been carefully defined so as to be independent of some syntactical choices. In particular, the order in which the types of significant fields have been declared leaves the meaning of rows unchanged. This is formally stated by the following Lemma.

1.11.9 Lemma: The equations of Figure 1-17 are equivalent to true. □

Proof: Each equation can be considered independently. It suffices to see that any ground assignment ϕ sends both sides of the equation to the same element

$$(\ell_1 : \mathtt{T}_1 \,;\, \ell_2 : \mathtt{T}_2 \,;\, \mathtt{T}_3) = (\ell_2 : \mathtt{T}_2 \,;\, \ell_1 : \mathtt{T}_1 \,;\, \mathtt{T}_3) \qquad \text{(C-Row-LL)}$$

$$\partial\mathtt{T} = (\ell : \mathtt{T} \,;\, \partial\mathtt{T}) \qquad \text{(C-Row-DL)}$$

$$\partial(G\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n) = G\ \partial\mathtt{T}_1\ \ldots\ \partial\mathtt{T}_n \qquad \text{(C-Row-DG)}$$

$$G\ (\ell : \mathtt{T}_1 \,;\, \mathtt{T}'_1)\ \ldots\ (\ell : \mathtt{T}_n \,;\, \mathtt{T}'_n) = (\ell : G\ \mathtt{T}_1\ \ldots\ \mathtt{T}_n \,;\, G\ \mathtt{T}'_1\ \ldots\ \mathtt{T}'_n) \qquad \text{(C-Row-GL)}$$

Figure 1-17: Equational reasoning for row types.

$$(\ell_1 : \mathtt{T}_1 \,;\, \mathtt{T}'_1) = (\ell_2 : \mathtt{T}_2 \,;\, \mathtt{T}'_2) \quad\equiv\quad \exists\mathtt{X}.(\mathtt{T}'_1 = (\ell_2 : \mathtt{T}_2 \,;\, \mathtt{X}) \wedge \mathtt{T}'_2 = (\ell_1 : \mathtt{T}_1 \,;\, \mathtt{X})) \qquad \text{(C-Mute-LL)}$$
$$\text{if } \mathtt{X} \# \mathit{ftv}(\mathtt{T}_1, \mathtt{T}'_1, \mathtt{T}_2, \mathtt{T}'_2) \wedge \ell_1 \neq \ell_2$$

$$(\ell : \mathtt{T} \,;\, \mathtt{T}') = G\ \mathtt{T}_i^I \quad\equiv\quad \exists(\mathtt{X}_i, \mathtt{X}'_i)^I.(\mathtt{T} = G\ \mathtt{X}_i^I \wedge \mathtt{T}' = G\ \mathtt{X}'^I_i \wedge \mathtt{T}_i = (\ell : \mathtt{X}_i \,;\, \mathtt{X}'_i))$$
$$\text{if } (\mathtt{X}_i, \mathtt{X}_i)^I \# \mathit{ftv}(\mathtt{T}, \mathtt{T}', \mathtt{T}_i^I) \qquad \text{(C-Mute-LG)}$$

$$\partial\mathtt{T} = G\ \mathtt{T}_i^I \quad\equiv\quad \exists\mathtt{X}_i^I.(\mathtt{T} = G\ \mathtt{X}_i^I \wedge (\mathtt{T}_i = \partial\mathtt{X}_i)^I) \qquad \text{(C-Mute-DG)}$$
$$\text{if } \mathtt{X}_i^I \# \mathit{ftv}(\mathtt{T}, \mathtt{T}_i^I)$$

$$\partial\mathtt{T} = (\ell : \mathtt{T}' \,;\, \mathtt{T}'') \quad\equiv\quad \mathtt{T} = \mathtt{T}' \wedge \partial\mathtt{T} = \mathtt{T}'' \qquad \text{(C-Mute-DL)}$$

Figure 1-18: Constraint equivalence laws for rows.

in the model, which follows directly from the meaning of row types. Notice that this fact only depends on the semantics of types and not on the semantics of the subtyping predicate. □

It follows from those equations that type constructors ℓ, ∂, and G are never isolated, each equation exhibiting a pair of compatible top symbols. Variances and incompatible pairs of type constructors depend on the signature $\mathcal{S}_0 \uplus \mathcal{S}_1$. However, it is not difficult to see that type constructors ∂ and ℓ are logically covariant in all directions and that the logical variance of types constructors G of $\mathit{dom}(\mathcal{S}_0 \uplus \mathcal{S}_1)$ correspond to their syntactic variance, which, in most cases, will allow the decomposition of equations with the same top symbols. Moreover, an equation between two terms whose top symbols form one of the four compatible pairs derived from the equations of Figure 1-17 holds only if immediate subexpressions can be "conciliated" in some way. There is a transformation quite similar to decomposition, called *mutation*, that mimics the equations for rows (Figure 1-17) and described by the rules of Figure 1-18. For sake of readability and conciseness, we write $\mathtt{T}_i^I$ instead of $\mathtt{T}_i^{i\in I}$.

1.11.10 Lemma [Mutation]: All equivalence laws in Figure 1-18 hold. □

Proof:

∘ *Case* C-Mute-LL: Let $\mathtt{X} \# \mathit{ftv}(\mathtt{T}_1, \mathtt{T}'_1, \mathtt{T}_2, \mathtt{T}'_2)$ **(1)** and $\ell_1 \neq \ell_2$. Let $\mathit{Row}(L)$ be the row kind of this equation. Let ϕ be a ground assignment that validates the constraint $(\ell_1 : \mathtt{T}_1 \; ; \; \mathtt{T}'_1) = (\ell_2 : \mathtt{T}_2 \; ; \; \mathtt{T}'_2)$. That is, ϕ sends all terms of the multi-equation to the same ground type t of row kind $\mathit{Row}(L)$. Moreover, the row-term semantics implies that t satisfies $t(\epsilon) = L$, $t/\ell_1 = \phi(\mathtt{T}_1) = \phi(\mathtt{T}'_2)/\ell_1$, $t/\ell_2 = \phi(\mathtt{T}'_1)/\ell_2 = \phi(\mathtt{T}_2)$, and $t/\ell = \phi(\mathtt{T}'_2)/\ell = \phi(\mathtt{T}'_1)/\ell$ for all $\ell \in \mathcal{L} \setminus \ell_1.\ell_2.L$ **(2)**. Let t' be the tree defined by $t'(\epsilon) = \ell_1.\ell_2.L$ and $t'/\ell = t/\ell$ for all $\ell \in \mathcal{L} \setminus \ell_1.\ell_2.L$. By construction and (2), $\phi[\mathtt{X} \mapsto t']$ satisfies both equations $\mathtt{T}'_1 = (\ell_2 : \mathtt{T}_2 \; ; \mathtt{X})$ and $\mathtt{T}'_2 = (\ell_1 : \mathtt{T}_1 \; ; \mathtt{X})$. Thus by CM-Exists and (1), ϕ satisfies $\exists \mathtt{X}.\mathtt{T}'_1 = (\ell_2 : \mathtt{T}_2 \; ; \mathtt{X}) \wedge \mathtt{T}'_2 = (\ell_1 : \mathtt{T}_1 \; ; \mathtt{X})$. Conversely, we have the entailment:

$$\begin{aligned}
& \exists \mathtt{X}.(\mathtt{T}'_1 = (\ell_2 : \mathtt{T}_2 \; ; \mathtt{X}) \wedge \mathtt{T}'_2 = (\ell_1 : \mathtt{T}_1 \; ; \mathtt{X})) \\
\equiv\ & \exists \mathtt{X}.((\ell_1 : \mathtt{T}_1 \; ; \mathtt{T}'_1) = (\ell_1 : \mathtt{T}_1 \; ; \ell_2 : \mathtt{T}_2 \; ; \mathtt{X}) \wedge \\
& \quad (\ell_2 : \mathtt{T}_2 \; ; \mathtt{T}'_2) = (\ell_2 : \mathtt{T}_2 \; ; \ell_1 : \mathtt{T}_1 \; ; \mathtt{X})) && \mathbf{(3)} \\
\Vdash\ & \exists \mathtt{X}.(\ell_1 : \mathtt{T}_1 \; ; \mathtt{T}'_1) = (\ell_2 : \mathtt{T}_2 \; ; \mathtt{T}'_2) && \mathbf{(4)} \\
\equiv\ & (\ell_1 : \mathtt{T}_1 \; ; \mathtt{T}'_1) = (\ell_2 : \mathtt{T}_2 \; ; \mathtt{T}'_2) && \mathbf{(5)}
\end{aligned}$$

(3) follows by covariance of ℓ_1 and ℓ_2; (4) by C-Row-LL and transitivity of equivalence; (5) by C-Ex* and (1).

∘ *Cases* C-Mute-LG, C-Mute-DG, and C-Mute-DL: The reasoning is similar to the case C-Mute-LL. □

Solving row constraints in an equality model

In this section, we extend the constraint solver for the equality-only free tree model (Figure 1-11), so as to handle rows. We thus assume an equality-only model.

Mutation is a common technique to solve equations in a large class of non-free algebras that are described by *syntactic theories* (Kirchner and Klay, 1990). The equations of Figure 1-17 happen to be a syntactic presentation of an equational theory, from which a unification algorithm could be automatically derived (Rémy, 1993). We recover the same transformation rules directly, without using results on syntactic theories.

The following lemma shows that all pairs of distinct type constructors for which there is no mutation rule are in fact incompatible.

1.11.11 Lemma: All symbols $H \in \mathcal{S}_1$ are isolated. Furthermore for every pair of distinct type constructors $G_1, G_2 \in \mathit{dom}(\mathcal{S}_0 \uplus \mathcal{S}_1)$, and every row kind s, we have $G_1^s \bowtie G_2^s$. □

Proof: Terms of the form $H\vec{\mathtt{T}}$ are interpreted by ground types with H at occurrence ϵ, and conversely the only interpretations of types with H at occurrence ϵ are terms of the form $H\vec{\mathtt{T}}$. Hence, no ground assignment can ever

$$\begin{array}{rcll}
(\ell_1 : \mathtt{X}_1 \;;\; \mathtt{X}'_1) = (\ell_2 : \mathtt{T}_2 \;;\; \mathtt{T}'_2) = \epsilon & \rightarrow & \exists \mathtt{Y}.(\mathtt{X}'_1 = (\ell_2 : \mathtt{T}_2 \;;\; \mathtt{Y}) \wedge \mathtt{T}'_2 = (\ell_1 : \mathtt{X}_1 \;;\; \mathtt{Y})) & \\
 & & \wedge\, (\ell_1 : \mathtt{X}_1 \;;\; \mathtt{X}'_1) = \epsilon & \text{(S-Mute-LL)} \\
 & & \text{if } \mathtt{Y} \# \mathit{ftv}(\mathtt{X}_1, \mathtt{X}'_1, \mathtt{T}_2, \mathtt{T}'_2) \wedge \ell_1 \neq \ell_2 & \\
(\ell : \mathtt{X} \;;\; \mathtt{X}') = G\; \mathtt{T}_i^{i \in I} = \epsilon & \rightarrow & \exists (\mathtt{Y}_i, \mathtt{Y}'_i)^{i \in I}.(\mathtt{X} = G\; \mathtt{Y}_i^{i \in I} \wedge \mathtt{X}' = G\; \mathtt{Y'}_i^{i \in I} \wedge \mathtt{T}_i = (\ell : \mathtt{Y}_i \;;\; \mathtt{Y}'_i)) & \\
 & & \wedge\, (\ell : \mathtt{X} \;;\; \mathtt{X}') = \epsilon & \text{(S-Mute-LG)} \\
 & & \text{if } (\mathtt{Y}_i, \mathtt{Y}'_i)^{i \in I} \# \mathit{ftv}(\mathtt{X}, \mathtt{X}', \mathtt{T}_i^{i \in I}) & \\
\partial \mathtt{X} = G\; \mathtt{T}_i^{i \in I} = \epsilon & \rightarrow & \exists \mathtt{Y}_i^{i \in I}.(\mathtt{X} = G\; \mathtt{Y}_i^{i \in I} \wedge (\mathtt{T}_i = \partial \mathtt{Y}_i)^{i \in I}) & \\
 & & \wedge\, \partial \mathtt{X} = \epsilon & \text{(S-Mute-DG)} \\
 & & \text{if } \mathtt{Y}_i^{i \in I} \# \mathit{ftv}(\mathtt{X}, \mathtt{T}_i^{i \in I}) & \\
\partial \mathtt{X} = (\ell : \mathtt{T} \;;\; \mathtt{T}') = \epsilon & \rightarrow & \mathtt{X} = \mathtt{T} \wedge \partial \mathtt{X} = \mathtt{T}' \wedge \partial \mathtt{X} = \epsilon & \text{(S-Mute-DL)} \\
G\, \vec{\mathtt{T}} = G'\, \vec{\mathtt{T}}' = \epsilon & \rightarrow & \mathsf{false} & \text{(S-Clash-I)} \\
 & & \text{if } F \bowtie F' &
\end{array}$$

Figure 1-19: Unification addendum for row types

send $H\vec{\mathtt{T}}$ and $F\vec{\mathtt{T}}'$ to the same ground term when $F \neq H$ and, as a result, H is isolated.

Let G_1 and G_2 be two type constructors of $\mathcal{S}_0$. For $s = \mathit{Type}$, the interpretations of terms of the form $G_1^s\vec{\mathtt{T}}$ and $G_2^s\vec{\mathtt{T}}'$ are ground types with symbols G_1 and G_2 at occurrence ϵ, respectively. Hence they cannot be made equal under any ground assignment. For $s = \mathit{Row}(L)$, the interpretations of terms of the form $G_1^s\vec{\mathtt{T}}$ and $G_2^s\vec{\mathtt{T}}'$ are ground types with constructor L at occurrence ϵ and constructors G_1 and G_2 at occurrence 1, respectively. Hence they cannot be made equal under any ground assignment. □

Any other combination of type constructors forms a compatible pair, as illustrated by equations of Figure 1-17 and can be transformed by mutation rules of Figure 1-18. The constraint solver for row-terms is the relation $\rightarrow^\dagger$ defined by the rewriting rules of Figure 1-11, except Rule S-Clash, which is replaced by the set of rules of Figure 1-19.

1.11.12 Lemma: The rewriting system $\rightarrow^\dagger$ is strongly normalizing. □

Please, note that the termination of $\rightarrow^\dagger$ relies on types being well-kinded. In particular, $\rightarrow^\dagger$ would not terminate on the ill-kinded system of equations $\mathtt{X} = \ell : \mathtt{T} \;;\; \mathtt{X}' \wedge \mathtt{X}' = \ell' : \mathtt{T} \;;\; \mathtt{X}$.

1.11.13 Lemma: If $U \rightarrow^\dagger U'$, then $U \equiv U'$. □

Proof: It suffices to check the property independently for each rule defining $\rightarrow^\dagger$. The proof for rules of Figure 1-11 but S-Clash remain valid for row terms. For S-Decompose, it follows by the invariance of all type constructors, which is preserved for row terms. For rule S-Class-I it follows by Lemma 1.11.11 and for mutation rules, it follows by Lemma 1.11.10. □

Although reductions $\rightarrow$ are not sound for row types, hence $\rightarrow$ cannot be used for computation over row types, it has some interest. In particular, the following property shows that normal forms for row types are identical to normal forms for regular types.

1.11.14 Lemma: A system U in normal form for $\rightarrow^\dagger$ is also in normal form for $\rightarrow$. □

Proof: The only rule of $\rightarrow$ that is not in $\rightarrow^\dagger$ is S-Clash. Thus, it suffices to observe that if Rule S-Clash would be applicable, then either Rule S-Class-I or a mutation rule would be applicable as well. □

As a corollary, Lemma 1.8.6 extends to row types.

Operations on records

We now illustrate the use of rows for typechecking operations on records. The simplest application of rows are full records with infinite carrier. Records types are expressed with rows instead of a simple product type. The basic operations are the same as in the finite case, that is, *creation*, *polymorphic update*, and *polymorphic access*, but labels are now taken from an infinite set. However, creation and polymorphic update, which where destructors are now taken as constructors and used to represent records as association lists. The access of a record v at a field ℓ is obtained by linearly searching v for a definition of field ℓ and returning this definition, or returning the default value if no definition has been found for ℓ.

1.11.15 Example [Full records]: We assume a unique basic kind $\star$ and a unique covariant isolated type constructor Π in $\mathcal{S}_1$. Let $\{\cdot\}$ be a unary constructor, $(\{\cdot\ \mathsf{with}\ \cdot = \ell\})^{\ell\in\mathcal{L}}$ a collection of binary constructors, and $(\ell.\{\cdot\})^{\ell\in\mathcal{L}}$ a collection of unary destructors with the following reduction rules:

$$
\begin{array}{rcll}
\{\mathtt{v}\}.\{\ell\} & \xrightarrow{\delta} & \mathtt{v} & & \text{(RD-Default)} \\
\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}.\{\ell\} & \xrightarrow{\delta} & \mathtt{v} & & \text{(RD-Found)} \\
\{\mathtt{w}\ \mathsf{with}\ \ell' = \mathtt{v}\}.\{\ell\} & \xrightarrow{\delta} & \mathtt{w}.\{\ell\} & \text{if } \ell \neq \ell' & \text{(RD-Follow)}
\end{array}
$$

Let the initial environment Γ_0 contain the following bindings

$$
\begin{array}{rl}
\{\cdot\} : & \forall \mathtt{X}.\ \mathtt{X} \rightarrow \Pi(\partial \mathtt{X}) \\
\{\cdot\ \mathsf{with}\ \ell = \cdot\} : & \forall \mathtt{X}\mathtt{X}'\mathtt{Y}.\ \Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \rightarrow \mathtt{X}' \rightarrow \Pi(\ell : \mathtt{X}'\ ;\ \mathtt{Y}) \\
\cdot.\{\ell\} : & \forall \mathtt{X}\mathtt{Y}.\ \Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \rightarrow \mathtt{X}
\end{array}
$$

□

1.11.16 Exercise [Full records, ★★★, $\nrightarrow$]: Check that these definitions meet the requirements of Definition 1.7.6. □

1.11.17 Exercise [Field Exchange, ★★]: Add an operation to permute two fields of a record: give the reduction rules and the typing assumptions and check that the requirements of Definition 1.7.6 are preserved. □

1.11.18 Exercise [Normal forms for records, ★★★]: Record values may contain repetitions. For instance, $\{\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}\ \mathsf{with}\ \ell = \mathtt{v}'\}$ is a value that is in fact observationally equivalent to $\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}'\}$. So are the two record values $\{\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}\ \mathsf{with}\ \ell' = \mathtt{v}'\}$ and $\{\{\mathtt{w}\ \mathsf{with}\ \ell' = \mathtt{v}'\}\ \mathsf{with}\ \ell = \mathtt{v}\}$ when $\ell' \neq \ell$. Modify the semantics, so that two record values of the same type have similar structure and records do not contain inaccessible values. □

1.11.19 Exercise [Map Apply, ★★]: Add a binary operation map such that the expressions $(\mathsf{map}\ \mathtt{v}\ \mathtt{w}).\{\ell\}$ and $\mathtt{v}.\{\ell\}\ \mathtt{w}.\{\ell\}$ reduce to the same value for every label ℓ. □

1.11.20 Exercise [★, $\nrightarrow$]: So far, full records are almost constants. This condition is not necessary for values, but only for types. As an example, introduce a primitive record, that is a nullary record constructor, that maps every label to a distinct integer. Give its typing assumption and review the semantics of records. □

As opposed to full records, standard records are partial and their domains are finite (but with infinite carrier) and statically determined from their types. Standard records can be built by extending the empty record on a finite number of fields. We refer to such records as records with *polymorphic extension*. Record with polymorphic extension can be obtained by means of encoding into full records, much as in the finite case.

1.11.21 Example [Encoding of polymorphic extension]: Reusing the two type definitions abs and pre that have introduced for the finite case, we let abs encodes an undefined field and $\mathsf{pre}\mathtt{v}$ encode a field with value $\mathtt{v}$. We use the following syntactic sugar with their meaning and principal types:

$$\begin{aligned}
\langle\rangle &\overset{\text{def}}{=} \{\mathsf{abs}\} \\
&\quad : \Pi(\partial\mathsf{abs}) \\
\langle\cdot\ \mathsf{with}\ \ell = \cdot\rangle &\overset{\text{def}}{=} \lambda\mathtt{v}.\lambda\mathtt{w}.\{\mathtt{w}\ \mathsf{with}\ \ell = \mathsf{pre}\ \mathtt{v}\} \\
&\quad : \forall \mathtt{X}\mathtt{X}'\mathtt{Y}.\ \Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \to \mathtt{X}' \to \Pi(\ell : \mathsf{pre}\ \ \mathtt{X}'\ ;\ \mathtt{Y}) \\
\cdot.\langle\ell\rangle &\overset{\text{def}}{=} \lambda\mathtt{v}.\mathsf{pre}^{-1}\ (\mathtt{v}.\{\ell\}) \\
&\quad : \forall \mathtt{X}\mathtt{Y}.\ \Pi(\ell : \mathsf{pre}\ \ \mathtt{X}\ ;\ \mathtt{Y}) \to \mathtt{X}
\end{aligned}$$

$\square$

1.11.22 EXERCISE [RECOMMENDED, ★]: Extension may actually override an existing field. Can you define a version polymorphic extension that prevents this situation from happening? Add an operation that hide some particular field of a record. $\square$

Extensible records can also be implemented directly, without encoding into full records. In fact, this requires only a tiny variation on full records.

1.11.23 EXAMPLE [RECORDS WITH POLYMORPHIC EXTENSION]: Let $\star$ and $\diamond$ be two basic kinds. Let the basic signature $\mathcal{S}_0$ contain (in addition to $\rightarrow$) the covariant isolated type constructors pre of kind $\star \Rightarrow \diamond$ and abs of kind $\diamond$. In the presence of subtyping, we may assume $\mathsf{pre} \leqslant \mathsf{abs}$. Let $\mathcal{S}_1$ contain the unique covariant isolated type constructor Π of kind $\diamond \Rightarrow \star$. Let $\langle\rangle$ be a unary constructor, $(\{\cdot \text{ with } \cdot = \ell\})^{\ell \in \mathcal{L}}$ be a binary constructor, and $(\ell.\{\cdot\})^{\ell \in \mathcal{L}}$ be a unary destructor with the following reduction rules:

$$\langle \mathtt{w} \text{ with } \ell = \mathtt{v}\rangle.\langle\ell\rangle \xrightarrow{\delta} \mathtt{v} \qquad \text{(ER-FOUND)}$$

$$\langle \mathtt{w} \text{ with } \ell' = \mathtt{v}\rangle.\langle\ell\rangle \xrightarrow{\delta} \mathtt{w}.\{\ell\} \qquad \text{if } \ell \neq \ell' \qquad \text{(ER-FOLLOW)}$$

Let Γ_0 contain the following typing assumptions:

$$\begin{aligned} \langle\rangle &: \Pi(\partial\mathsf{abs}) \\ \langle \cdot \text{ with } \ell = \cdot\rangle &: \forall \mathtt{X}\mathtt{X}'\mathtt{Y}.\, \Pi(\ell : \mathtt{X} \,;\, \mathtt{Y}) \rightarrow \mathtt{X}' \rightarrow \Pi(\ell : \mathsf{pre}\ \mathtt{X}' \,;\, \mathtt{Y}) \\ \cdot.\langle\ell\rangle &: \forall \mathtt{X}\mathtt{Y}.\, \Pi(\ell : \mathsf{pre}\ \mathtt{X} \,;\, \mathtt{Y}) \rightarrow \mathtt{X} \end{aligned}$$

$\square$

Notice that the typing assumptions obtained from the direct approach are identical to those obtained via the encoding into full records in Example 1.11.21.

1.11.24 EXERCISE [★★★★, $\nrightarrow$]: Prove the equivalence between the direct semantics and the semantics via the encoding into records with a default. $\square$

1.11.25 EXERCISE [RECOMMENDED, ★★, $\nrightarrow$]: Prove that type soundness for extensible records hold in both the subtyping model and equality-only models. $\square$

1.11.26 EXERCISE [RECOMMENDED, ★, $\nrightarrow$]: Check that in the subtyping a record with more fields can be used in place of records with fewer fields. Check that this is not the case in the equality-only model. $\square$

1.11.27 Example [Refinement of record types]: In an equality-only model, records with more fields cannot be used in place of records with fewer fields. However, this may be partially recovered by a small refinement of the structure of types. The presence of fields can actually be split form their types, thus enabling some polymorphism over the presence of fields while type of fields themselves remains fixed. Let $\circ$ be a new basic kind. Let type constructors abs and pre be both of kind $\circ$ and let $\cdot$ be a new type constructor of kind $\circ \otimes \star \Rightarrow \diamond$. Let Γ_0 contain the following typing assumptions:

$$\begin{aligned}
\langle\rangle : &\quad \forall \mathtt{X}.\Pi(\partial(\mathsf{abs}\cdot\mathtt{X}))\\
\langle\cdot \;\mathsf{with}\; \ell = \cdot\rangle : &\quad \forall \mathtt{Z}\mathtt{X}\mathtt{X}'\mathtt{Y}.\, \Pi(\ell : \mathtt{X} \;;\; \mathtt{Y}) \to \mathtt{X}' \to \Pi(\ell : \mathtt{Z}\cdot\mathtt{X}' \;;\; \mathtt{Y})\\
\cdot.\langle\ell\rangle : &\quad \forall \mathtt{X}\mathtt{Y}.\, \Pi(\ell : \mathsf{pre}\cdot\mathtt{X} \;;\; \mathtt{Y}) \to \mathtt{X}
\end{aligned}$$

The semantics of records remain unchanged. The new signature strictly generalizes the previous one (strictly more programs can be typed) while preserving type soundness. Here is a program that can now be typed and that could not be typed before:

$$(\mathtt{if}\ a\ \mathtt{then}\ \langle\langle\langle\rangle\ \mathsf{with}\ \ell' = \mathtt{true}\rangle\ \mathsf{with}\ \ell = 1\rangle\ \mathtt{else}\ \langle\langle\rangle\ \mathsf{with}\ \ell = 2\rangle).\langle\ell\rangle$$

Notice however, that when a present field is forgotten, the type of the field is not. Therefore two records defining the same field but with incompatible types can still not be mixed, which is possible in the subtyping model. □

1.11.28 Example [Refined subtyping]: The previous refinement for an equality-only model is not much interesting in the case of a subtyping model.

The subtyping assumption $\mathsf{pre} \leqslant \mathsf{abs}$ makes abs play the role of $\top$ for fields. That is, abs encodes the absence of information and not the information of absence. In other words, a value whose field ℓ has type abs may either be undefined or defined on field ℓ; in the latter case, the fact that field ℓ is actually defined has just been forgotten. Thus, types only provides a lower approximation of the actual domain of records. This is a lost of accuracy by comparison with the equality-only model, where a record domain is known from its type. As a result, some optimizations in the representation of records that are only possible when the exact domain of a record is statically known are lost.

Fortunately, there is a way to recover such accuracy. A conservative solution could of course to drop the inequality $\mathsf{pre} \leqslant \mathsf{abs}$. Notice that this would still be more expressive than using an equality model since, for instance $\Pi(\ell : \mathsf{pre}\ (\mathtt{T}_1 \to \mathtt{T}_2) \;;\; \mathtt{T}) \leq \Pi(\ell : \mathsf{pre}\ \top \;;\; \mathtt{T})$ would still hold, as long as $\to\, \leq \top$ does hold. This solution is known as depth-only subtyping for records, while the previous one provided both depth and width record subtyping. Conversely, one could also keep width subtyping and disallow depth subtyping, by preserving the relation $\mathsf{pre} \leqslant \mathsf{abs}$ while requiring pre to be invariant; in this case, presence of fields can be forgotten as a whole, but the types of fields cannot be weakened as long as fields remain visible.

Another more interesting solution consists in introducing another type constructor either of signature $\diamond$ and assuming that $\mathsf{pre} \leqslant \mathsf{either}$ and $\mathsf{abs} \leqslant \mathsf{either}$ (but $\mathsf{pre} \not\leqslant \mathsf{abs}$). Here, either plays the role of $\top$ for fields and means either *present* (and forgotten) or *absent*. while abs really means *absent*. The accuracy of typechecking can be formally stated as the fact that a record value of type $\Pi(\ell : \mathsf{abs} \,;\, \mathtt{T})$ cannot define field ℓ. □

1.11.29 Example [mixed subtyping]: It is tempting to mix all variations of Example 1.11.28 together. As a first attempt, we may assume that the basic signature $\mathcal{S}_0$ contains covariant type constructors pre and maybe and invariant type constructors $\mathsf{pre}_=$ and $\mathsf{maybe}_=$, all of kind $\star \Rightarrow \diamond$ and two type constructors abs and either of kind $\diamond$, and that the subtype ordering $\leqslant$ is defined by the following diagram:

$$\begin{tikzcd}
 & \mathsf{either} & & \\
 & \mathsf{maybe} \arrow[u] & & \\
\mathsf{pre} \arrow[ur] & & \mathsf{maybe}_= \arrow[ul] & \\
 & \mathsf{pre}_= \arrow[ul] \arrow[ur] & & \mathsf{abs} \arrow[ul]
\end{tikzcd}$$

Intuitively, we wish that $\mathsf{pre}_=$ and $\mathsf{maybe}_=$ be *logically* invariant, pre and maybe be *logically* covariant, and the equivalences $\mathsf{pre}_=\ \mathtt{T} \leq \mathsf{maybe}_=\ \mathtt{T}' \equiv \mathtt{T} = \mathtt{T}'$ and

$$\mathsf{pre}_=\ \mathtt{T} \leq \mathsf{pre}\ \mathtt{T}' \equiv \mathsf{pre}\ \mathtt{T} \leq \mathsf{maybe}\ \mathtt{T}' \equiv \mathsf{maybe}_=\ \mathtt{T} \leq \mathsf{maybe}\ \mathtt{T}' \equiv \mathtt{T} \leq \mathtt{T}' \qquad (\mathbf{1})$$

simultaneously hold. However, (1) requires, for instance, type constructors $\mathsf{pre}_=$ and pre to have the same direction, which is not currently possible since they do not have the same variance. Interestingly, this restriction may be relaxed by assigning variances of directions on a per type constructor basis and define structural subtyping accordingly (See Exercise 1.11.30). Then, replacing all occurrences of pre by $\mathsf{pre}_=$ in Γ_0 preserves type soundness and allows for both accurate record types and flexible subtyping in the same setting. □

1.11.30 Exercise [Relaxed variances, ★★★, $\nrightarrow$]: Let $\emptyset$ be allowed as a new variance, let extend the composition of variances defined in Example 1.3.9 with $\nu\emptyset = \emptyset$, and let $\leqslant^{\emptyset}$ stands for the full relation on type constructors. Let each type constructor F of signature $K \Rightarrow \kappa$ now come with a mapping $\vartheta(F)$ from $dom(K)$ to variances. Let $\vartheta(t, t', \pi)$ be the variance of two ground types t and t' at a path π recursively defined by $\vartheta(t, t', d \cdot \pi) = \big(\vartheta(t(\epsilon))(d) \cap \vartheta(t'(\epsilon))(d)\big)\ \vartheta(t/d, t'/d, \pi)$ and $\vartheta(t, t', \epsilon) = +$. Then define the interpretation of subtyping as follows: if $t, t' \in \mathcal{M}_\kappa$, let $t \leq t'$ hold if and only if for all path $\pi \in dom(t) \cap dom(t')$, $t(\pi) \leqslant^{\vartheta(t,t',\pi)} t'(\pi)$ holds.

Check that the relation $\leq$ remains a partial ordering. Check that a type constructor whose direction d has been syntactically declared covariant (respectively contravariant, invariant) is still logically covariant (respectively contravariant, invariant) in d. □

Record concatenation

Record concatenation takes two records and combines them into a new record whose fields are taken from whatever argument defines them. Of course, there is an ambiguity when the two records do not have disjoint domains and a choice should be made to disambiguate such cases. *Symmetric* concatenation let concatenation be undefined in this case (Harper and Pierce, 1991), while *asymmetric* concatenation let one-side (usually the right side) always take priority. Despite a rather simple semantics, record concatenation remains hard to type (with either a strict or a priority semantics). Solutions to type inference for record concatenation may be found, for instance, in (Wand, 1989; Rémy, 1992; Pottier, 2000).

Polymorphic variants

Variants can be defined via algebraic data-type definitions. However, as fields for records, variant tags are taken from a relatively small, finite collection of labels and two variant definitions will have incompatible types. Thus, to remain compatible, two variants must chose their tag among a larger collection that is a superset of all the possible tags of either variant. In general, this reduces the accuracy of types and forces useless dynamic checks for tags that could otherwise be known not to occur. Extensible variants (page 93) allow to work with an arbitrary large collection of tags, but do not improve accuracy. Polymorphic variants refers to a more precise typechecking mechanism for variants, where types more accurately describes the tags that may actually occur. They allow to build values of sum types out of a large, potentially infinite predefined set of tags and call polymorphic functions to explore them. As for record, this problem could be tackled by first considering polymorphic operations over variants built from a finite set of tags and total variants with an infinite set of tag independently and then by combining both approaches together. We propose a direct solution by a simple analogy with records.

Indeed, type constructor pre can be used to distinguish a (finite) set of tags that the variant may actually carry, from other tags that are certain not to occur and typed with abs. For example, a variant $\ell.\mathtt{v}$, built from a value $\mathtt{v}$ with a constructor tag ℓ of arity one. may be assigned the principal type scheme $\forall\mathtt{X}.\Sigma(\ell : \mathsf{pre}\ \mathtt{T}; \mathtt{X})$ where $\mathtt{T}$ is the type of $\mathtt{v}$. The unary type constructor Σ is used to coerce rows to variant types—thus, variant types and record types may share the same inner row structure and be simply distinguished by their top symbol. An instance of this polymorphic type is $\forall\mathtt{X}.\Sigma(\ell{:}\mathsf{pre}\ \mathtt{T}\ ;\ \mathsf{abs})$, which tells that the variant must have been built with tag ℓ and no other tag, thus retaining exact information about the shape of the value. Another instance of the variant polymorphic type is $\Sigma(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \ell' : \mathsf{pre}\ \mathtt{T}'\ ;\ \mathsf{abs})$. Indeed, it is sound to assume that the value might also have been built with some other tag ℓ', even if we know that this is not actually the case. Interestingly, both values $\ell.\mathtt{v}$ and $\ell'.\mathtt{v}$ have this type and can be mixed at this type.

We use filters to explore variants. A filter $[\,\ell : \mathtt{v} \mid \mathtt{v}'\,]$ is a function that expects a variant argument, thus of the form $\ell'.\mathtt{w}$. It then proceeds with either $\mathtt{v}\ \mathtt{w}$, if $\ell' = \ell$, or $\mathtt{v}'\ \mathtt{w}$ otherwise. The type of this filter is $\Sigma(\ell : \mathsf{pre}\ \mathtt{T}\ ;\ \mathtt{T}') \to \mathtt{T}''$ where $\mathtt{T}$ is the type of values accepted by $\mathtt{v}$, $\Sigma(\ell : \mathtt{T}'''\ ;\ \mathtt{T}')$ is the type of values accepted by $\mathtt{v}'$, and $\mathtt{T}''$ is the type of values returned by both $\mathtt{v}$ and $\mathtt{v}'$. Any type $\mathtt{T}'''$ would do, including, in particular, abs. Indeed, when $\mathtt{w}$ is passed to $\mathtt{v}'$, it is known not to have tag ℓ, so the behavior of $\mathtt{v}'$ on ℓ does not matter. The null filter $[]$ can be used for $\mathtt{v}'$. This filter should actually never be applied, which we ensure by assigning $[]$ the type $\forall\mathtt{X}.\Sigma(\partial\mathsf{abs}) \to \mathtt{X}$, for no variant value has type $\Sigma(\partial\mathsf{abs})$. For instance, the filter $[\,\ell : \mathtt{v}_\ell \mid [\,\ell' : \mathtt{v}_{\ell'} \mid []\,]\,]$, which may be abbreviated as $[\,\ell : \mathtt{v}_\ell \mid \ell' : \mathtt{v}_{\ell'}\,]$ can be applied to either $\ell.\mathtt{v}$ or $\ell'.\mathtt{v}'$. The following example formalizes polymorphic variants.

1.11.31 Example [Polymorphic variants]: Let $\star$ and $\diamond$ be two basic kinds. Let $\mathcal{S}$ contain in addition to the arrow type constructor the two type constructors pre of kind $\star \Rightarrow \diamond$ and abs of kind $\diamond$. In the presence of subtyping we may assume $\mathsf{abs} \leqslant \mathsf{pre}$. Let $\mathcal{S}_1$ contain the unique covariant isolated type constructor Σ of kind $\diamond \Rightarrow \star$. Let Γ_0 be composed of unary constructors $(\ell.\cdot)^{\ell\in\mathcal{L}}$ and primitives $[]$ of arity 0 and $([\,\ell : \cdot \mid \cdot\,]\cdot)^{\ell\in\mathcal{L}}$ of arity 3, given with the following reduction rules:

$$\begin{array}{lcllr}
[\,\ell : \mathtt{v} \mid \mathtt{v}'\,]\ \ell.\mathtt{w} & \xrightarrow{\delta} & \mathtt{v}\ \mathtt{w} & & \text{(EV-FOUND)}\\
[\,\ell : \mathtt{v} \mid \mathtt{v}'\,]\ \ell'.\mathtt{w} & \xrightarrow{\delta} & \mathtt{v}'\ \mathtt{w} & \text{if } \ell \neq \ell' & \text{(EV-FOLLOW)}
\end{array}$$

and contain the following typing assumptions:

$$\begin{array}{rl}
\ell.\cdot : & \forall\mathtt{XY}.\ \mathtt{X} \to \Sigma(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y})\\
[] : & \forall\mathtt{X}.\Sigma(\partial\mathsf{abs}) \to \mathtt{X}\\
[\,\ell : \cdot \mid \cdot\,] : & \forall\mathtt{XX'YY'}.\ (\mathtt{X} \to \mathtt{Y}) \to (\Sigma(\ell : \mathtt{X}'\ ;\ \mathtt{Y}') \to \mathtt{Y}) \to \Sigma(\ell : \mathsf{pre}\ \mathtt{X}\ ;\ \mathtt{Y}') \to \mathtt{Y}
\end{array}$$

□

1.11.32 Exercise [Soundness for Extensible Variants, ★★★, $\not\rightarrow$]: Prove type soundness for extensible variants in both equality-only and subtyping models. □

Other applications of rows

Polymorphic records and variants are the most well-known applications of rows. Besides the many variations on their presentations—we have only illustrated some of them—there are several other interesting applications of rows.

Since objects can be viewed as record-of-functions, at least from a typing point of view, rows can also be used to type structural objects (Wand, 1994; Rémy, 1994; Rémy and Vouillon, 1998) and provide, in particular, polymorphic method invocation. This is the key to typechecking objects in Objective Caml (Rémy and Vouillon, 1998). First-class messages (Nishimura, 1998; Müller and Nishimura, 1998; Pottier, 2000) combine records and variants in an interesting way: while filters over variant types enforce all branches to have the same return type, first-class messages treat filters as records of functions (also called objects) rather than functions from a variant type to a shared return type. A message is an element of a variant type. The application of an object to a message, that is of a record of functions to a variant type, selects from the record the branch labeled with the same tag as the message and applies it to the content of the message, much as pattern matching. However, these applications are typechecked more accurately by first restricting the domain of the record to the set of tags that the message may possibly carry, and thus other branches and in particular their return type are left unconstrained.

Row types may also represent set of properties within types or type refinements and be used in type systems for program analysis. Two examples worth mentioning are their application to soft-typing (Cartwright and Fagan, 1991; Wright and Cartwright, 1994) and typechecking of uncaught exceptions (Leroy and Pessaux, 2000).

The key to rows is to decompose the set of row labels into a class of finite partitions that is closed by some operations. Here, those partitions are composed of singleton labels and co-finite sets of labels; the operations are merging (or conversely splitting) a singleton label and a co-finite set of labels. Other decompositions are possible, for instance, one could imagine to consider labels in a two-dimensional space. More generally, labels might also be given internal structure, for instance, one might consider automatons as labels. Notice also that record types are stratified, since rows, that is, expressions of kind $Row(L)$, may not themselves contain records —constructors of $\mathcal{S}_1$ are only given the image row kind *Type*. This restriction can be partially relaxed leading to rows of increasing degrees (Rémy, 1992b) ... and complexity! Yet more intriguing are typed-indexed rows where labels are themselves types (Shields and Meijer, 2001).

Alternatives to rows

The original idea of using rows to describe types of extensible records is due to Wand (Wand, 1987, 1988). A key simplification to row types is to make them total functions from labels to types and encode definiteness explicitly in the structure of fields, for instance with pre and abs type constructors, as presented here. This decomposition reduces the resolution of unification constraints to a simple equational reasoning (Rémy, 1993, 1992a). Other approaches that do not treat rows as total functions seem more *ad hoc* and have often hard-wired restrictions (Jategaonkar and Mitchell, 1988; Ohori and Buneman, 1989; Berthomieu, 1993; Ohori, 1999). Among these partial solutions, (Ohori, 1999) is quite interesting for its overall simplicity in the case where polymorphic access alone is required. Rows and fields may also be represented within ad-hoc type constraints rather than terms and equality (or subtyping) constraints. For example, qualified types use the predicates $(\mathtt{T}\ \mathsf{has}\ \ell : \mathtt{T}')$ and $(\mathtt{T}\ \mathsf{lacks}\ \ell)$ to mean that field ℓ of row $\mathtt{T}$ is defined with type $\mathtt{T}'$ or undefined, respectively (Jones, 1994b; Odersky, Sulzmann, and Wehr, 1999b). These constraints are in fact equivalent in our equality-model to $\exists \mathtt{X}.\mathtt{T} = (\ell : \mathsf{pre}\ \mathtt{T}' \,;\, \mathtt{X})$ and $\exists \mathtt{X}.\mathtt{T} = (\ell : \mathsf{abs} \,;\, \mathtt{X})$, respectively. Record typechecking has also been widely studied in the presence of subtyping. Usually, record subtyping is given meaning directly and not via rows. While these solutions are quite expressive, thanks to subtyping, they still suffer from their nonstructural treatment of record types and cannot type row extension. Thus, even in subtyping models the use of rows increases expressiveness, and is usually a simplification as well. The subtyping model can then also take advantage of the possibility of enriching type constructors pre and abs with more structure and relate them via subtyping (Pottier, 2000). Notice, that even though rows have been introduced for type inference, they seem to be beneficial to explicitly typed languages as well since even other advanced solutions (Cardelli and Mitchell, 1991; Cardelli, 1992) are limited.

Rules of Figure 1-19 are *one way* of solving row type constraints. In a model with subtyping constraints, a more direct closure-based resolution may be more appropriate (Pottier, 2003).

B *Solutions to Selected Exercises*

1.2.6 SOLUTION: The definition does not behave as expected, because `if` is a destructor, whose arguments—according to the call-by-value semantics of ML-the-calculus—are evaluated *before* R-TRUE or R-FALSE is allowed to fire. As a result, the semantics of the expression $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ is to evaluate *both* t_1 and t_2 before choosing one of them. Since these expressions may have side effects (for instance, they may fail to terminate, or update a reference), this semantics is undesirable. The desired evaluation order can be obtained by placing t_1 and t_2 within closures, which delays their evaluation, then invoking the closure returned by the conditional, forcing its body to be evaluated. In other words, the expression $\texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ should now be viewed as syntactic sugar for $\texttt{if}\ t_0\ (\lambda z.t_1)\ (\lambda z.t_2)\ \hat{0}$. The choice of the constant $\hat{0}$ is arbitrary, since it is discarded; any value would do.

1.2.21 SOLUTION: Within Damas and Milner's type system, we have:

$$\dfrac{\dfrac{\dfrac{}{z_1 : X \vdash z_1 : X}\ \textsc{dm-Var} \qquad \dfrac{}{z_1 : X; z_2 : X \vdash z_2 : X}\ \textsc{dm-Var}}{z_1 : X \vdash \texttt{let}\ z_2 = z_1\ \texttt{in}\ z_2 : X}\ \textsc{dm-Let}}{\varnothing \vdash \lambda z_1.\texttt{let}\ z_2 = z_1\ \texttt{in}\ z_2 : X \to X}\ \textsc{dm-Abs}$$

Please note that, because X occurs free within the environment $z_1 : X$, it is impossible to apply DM-GEN to the judgement $z_1 : X \vdash z_1 : X$ in a nontrivial way. For this reason, z_2 cannot receive the type scheme $\forall X.X$, and the whole expression cannot receive type $X \to Y$, where X and Y are distinct.

1.2.22 SOLUTION: It is straightforward to prove that the identity function has type $\texttt{int} \to \texttt{int}$:

$$\dfrac{\dfrac{}{\Gamma_0; z : \texttt{int} \vdash z : \texttt{int}}\ \textsc{dm-Var}}{\Gamma_0 \vdash \lambda z.z : \texttt{int} \to \texttt{int}}\ \textsc{dm-Abs}$$

In fact, nothing in this type derivation depends on the choice of `int` as the type of z. Thus, we may just as well use a type variable X instead. Furthermore, after forming the arrow type $X \to X$, we may employ DM-GEN to quantify universally over X, since it no longer appears in the environment.

$$\dfrac{\dfrac{\dfrac{}{\Gamma_0; z : X \vdash z : X}\ \textsc{dm-Var}}{\Gamma_0 \vdash \lambda z.z : X \to X}\ \textsc{dm-Abs} \qquad X \notin \mathit{ftv}(\Gamma_0)}{\Gamma_0 \vdash \lambda z.z : \forall X.X \to X}\ \textsc{dm-Gen}$$

It is worth noting that, although the type derivation employs an arbitrary type variable X, the final typing judgement has no free type variables. It is thus independent of the choice of $\mathtt{X}$. In the following, we refer to the above type derivation as Δ_0.

Next, we prove that the successor function has type $\mathtt{int} \rightarrow \mathtt{int}$ under the initial environment Γ_0. We write Γ_1 for $\Gamma_0; \mathtt{z} : \mathtt{int}$, and make uses of DM-VAR implicit.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Gamma_1 \vdash \hat{+} : \mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \\ \Gamma_1 \vdash \mathtt{z} : \mathtt{int}\end{array}}{\Gamma_1 \vdash \hat{+}\ \mathtt{z} : \mathtt{int} \rightarrow \mathtt{int}}\ \textsc{dm-App} \qquad \Gamma_1 \vdash \hat{1} : \mathtt{int}}{\Gamma_1 \vdash \mathtt{z} \mathbin{\hat{+}} \hat{1} : \mathtt{int}}\ \textsc{dm-App}}{\Gamma_0 \vdash \lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1} : \mathtt{int} \rightarrow \mathtt{int}}\ \textsc{dm-Abs}$$

In the following, we refer to the above type derivation as Δ_1. We may now build a derivation for the third typing judgement. We write Γ_2 for $\Gamma_0; \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}$.

$$\dfrac{\Delta_1 \qquad \dfrac{\Gamma_2 \vdash \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int} \qquad \Gamma_2 \vdash \hat{2} : \mathtt{int}}{\Gamma_2 \vdash \mathtt{f}\ \hat{2} : \mathtt{int}}\ \textsc{dm-App}}{\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{z} \mathbin{\hat{+}} \hat{1}\ \mathtt{in}\ \mathtt{f}\ \hat{2} : \mathtt{int}}\ \textsc{dm-Let}$$

To derive the fourth typing judgement, we re-use Δ_0, which proves that the identity function has polymorphic type. We write Γ_3 for $\Gamma_0; \mathtt{f} : \forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$. By DM-VAR and DM-INST, we have $\Gamma_3 \vdash \mathtt{f} : (\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (\mathtt{int} \rightarrow \mathtt{int})$ and $\Gamma_3 \vdash \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}$. Thus, we may build the following derivation:

$$\dfrac{\Delta_0 \qquad \dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\Gamma_3 \vdash \mathtt{f} : (\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (\mathtt{int} \rightarrow \mathtt{int}) \\ \Gamma_3 \vdash \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}\end{array}}{\Gamma_3 \vdash \mathtt{f}\ \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}}\ \textsc{dm-App} \\ \Gamma_3 \vdash \hat{2} : \mathtt{int}\end{array}}{\Gamma_3 \vdash \mathtt{f}\ \mathtt{f}\ \hat{2} : \mathtt{int}}\ \textsc{dm-App}}{\Gamma_0 \vdash \mathtt{let}\ \mathtt{f} = \lambda \mathtt{z}.\mathtt{z}\ \mathtt{in}\ \mathtt{f}\ \mathtt{f}\ \hat{2} : \mathtt{int}}\ \textsc{dm-Let}$$

The first and third judgements are valid in the simply-typed λ-calculus, because they use neither DM-GEN nor DM-INST, and use DM-LET only to introduce the *monomorphic* binding $\mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}$ into the environment. The second judgement, of course, is not: because it involves a nontrivial type scheme, it is not even a well-formed judgement in the simply-typed λ-calculus. The fourth judgement is well-formed, but *not* derivable, in the simply-typed λ-calculus. This is because $\mathtt{f}$ is used at two incompatible types, namely $(\mathtt{int} \rightarrow \mathtt{int}) \rightarrow (\mathtt{int} \rightarrow \mathtt{int})$ and $\mathtt{int} \rightarrow \mathtt{int}$, inside the expression $\mathtt{f}\ \mathtt{f}\ \hat{2}$. Both of these types are *instances* of $\forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$, the type scheme assigned to $\mathtt{f}$ in the environment Γ_3.

By inspection of DM-VAR, DM-GEN, and DM-INST, it is straightforward to see that, if $\Gamma_0 \vdash \hat{1} : \mathtt{T}$ is derivable, then $\mathtt{T}$ must be `int`. Since `int` is not an arrow type, the application $\hat{1}\,\hat{2}$ cannot be well-typed under Γ_0. In fact, because this expression is stuck, it cannot be well-typed in a sound type system.

The expression $\lambda\mathtt{f}.(\mathtt{f}\;\mathtt{f})$ is ill-typed in the simply-typed λ-calculus, because no type $\mathtt{T}$ may coincide with a type of the form $\mathtt{T} \rightarrow \mathtt{T}'$. Indeed, $\mathtt{T}$ would then be a subterm of itself. For the same reason, this expression is ill-typed in DM as well. Indeed, it is not difficult to check that the presence of DM-GEN and DM-INST makes no difference: DM-GEN cannot generalize $\mathtt{T}$ as long as the binding $\mathtt{f} : \mathtt{T}$ appears in the environment, and DM-INST can only instantiate $\mathtt{T}$ to $\mathtt{T}$ itself. Thus, the self-application $\mathtt{f}\;\mathtt{f}$ is well-typed in DM *only* if $\mathtt{f}$ is `let`-bound, as opposed to λ-bound. The argument crucially relies on the fact that $\mathtt{f}$ must be assigned a *monotype*. Indeed, the expression $\lambda\mathtt{f}.(\mathtt{f}\;\mathtt{f})$ is well-typed in an implicitly-typed variant of System F: one of its types is $(\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}) \rightarrow (\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X})$. It also relies on the fact that types are *finite*: indeed, this expression is well-typed in an extension of the simply-typed λ-calculus with recursive types, where the equation $\mathtt{T} = \mathtt{T} \rightarrow \mathtt{T}'$ has a solution.

1.2.23 SOLUTION: It is clear that the effect of DM-GEN may be obtained by a series of successive applications of DM-GEN'. Conversely, consider an instance of DM-GEN', whose premises are $\Gamma \vdash \mathtt{t} : \mathtt{S}$ **(1)** and $\mathtt{X} \notin \mathit{ftv}(\Gamma)$ **(2)**. Let us write $\mathtt{S} = \forall\bar{\mathtt{X}}.\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(\Gamma)$ **(3)**. Applying DM-INST to (1) and to the identity substitution yields $\Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. Applying DM-GEN to (4), (2) and (3) yields $\Gamma \vdash \mathtt{t} : \forall\mathtt{X}\bar{\mathtt{X}}.\mathtt{T}$, that is, $\Gamma \vdash \mathtt{t} : \forall\mathtt{X}.\mathtt{S}$. Thus, the effect of DM-GEN' may be obtained by DM-INST and DM-GEN.

It is clear that DM-INST is a particular case of DM-INST' where $\bar{\mathtt{Y}}$ is empty. Conversely, consider an instance of DM-INST', whose premises are $\Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{X}}.\mathtt{T}$ **(1)** and $\bar{\mathtt{Y}} \# \mathit{ftv}(\forall\bar{\mathtt{X}}.\mathtt{T})$ **(2)**. Let ρ be a renaming that exchanges $\bar{\mathtt{Y}}$ with $\bar{\mathtt{Z}}$, where $\bar{\mathtt{Z}} \# \mathit{ftv}(\forall\bar{\mathtt{Y}}.[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$ **(3)** and $\bar{\mathtt{Z}} \# \mathit{ftv}(\Gamma)$ **(4)**. Applying DM-INST to (1) yields $\Gamma \vdash \mathtt{t} : [\vec{\mathtt{X}} \mapsto \rho\vec{\mathtt{T}}]\mathtt{T}$ **(5)**. Applying DM-GEN to (5) and (4) yields $\Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{Z}}.[\vec{\mathtt{X}} \mapsto \rho\vec{\mathtt{T}}]\mathtt{T}$, that is, $\Gamma \vdash \mathtt{t} : \forall\rho\bar{\mathtt{Y}}.[\vec{\mathtt{X}} \mapsto \rho\vec{\mathtt{T}}]\mathtt{T}$ **(6)**. Now, by (2) and (3), we have $[\vec{\mathtt{X}} \mapsto \rho\vec{\mathtt{T}}]\mathtt{T} = \rho([\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$, so (6) may be written $\Gamma \vdash \mathtt{t} : \forall\rho\bar{\mathtt{Y}}.\rho([\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$, that is, $\Gamma \vdash \mathtt{t} : \rho(\forall\bar{\mathtt{Y}}.[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T})$ **(7)**. By (3), this is exactly $\Gamma \vdash \mathtt{t} : \forall\bar{\mathtt{Y}}.[\vec{\mathtt{X}} \mapsto \vec{\mathtt{T}}]\mathtt{T}$. Thus, the effect of DM-INST' may be obtained by DM-INST and DM-GEN.

1.4.4 SOLUTION: Let us recall that a program $\mathtt{t}$ is well-typed if and only if a judgement of the form $C, \Gamma \vdash \mathtt{t} : \sigma$, where C is satisfiable, holds. Let us show that it is in fact possible, without loss of generality, to require σ to be a monotype.

Assume $C, \Gamma \vdash \mathtt{t} : \sigma$ **(1)** is derivable within HM(X). Let us write $\sigma = \forall\bar{\mathtt{X}}[D].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(C)$ **(2)**. Applying Lemma 1.4.1 to (1) yields $C \Vdash \exists\bar{\mathtt{X}}.D$ **(3)**. By HM-INST, (1) implies $C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(4)**. By (3), we have $C \equiv C \wedge \exists\bar{\mathtt{X}}.D \equiv \exists\bar{\mathtt{X}}.(C \wedge D)$. Because C is satisfiable, this implies that $C \wedge D$

is satisfiable as well. Thus, the judgement (4), which involves the monotype $\mathtt{T}$, witnesses that $\mathtt{t}$ is well-typed.

We have shown that a program $\mathtt{t}$ is well-typed if and only if a judgement of the form $C, \Gamma \vdash \mathtt{t} : \mathtt{T}$, where C is satisfiable, holds. Thus, by Theorems **??** and **??**, well-typedness is the same for both rule sets.

1.4.5 Solution: By Theorem **??**, every rule in Figure 1-8 is admissible in HM(X). Of course, so is hm-Gen. So, every judgement that is derivable via the rules of Figure 1-8 and hm-Gen is a valid HM(X) judgement.

Conversely, assume $C, \Gamma \vdash \mathtt{t} : \sigma$ **(1)** holds in HM(X). We must show that it is derivable via the rules of Figure 1-8 and hm-Gen. Let us write $\sigma = \forall \bar{\mathtt{X}}[D].\mathtt{T}$, where $\bar{\mathtt{X}} \# \mathit{ftv}(C, \Gamma)$ **(2)**. By hm-Inst and (1), the judgement $C \wedge D, \Gamma \vdash \mathtt{t} : \mathtt{T}$ **(3)** holds in HM(X). This judgement involves a monotype, so, by Theorem **??**, it is derivable via the rules of Figure 1-8. Furthermore, from (3) and (2), hm-Gen allows deriving $C \wedge \exists \sigma, \Gamma \vdash \mathtt{t} : \sigma$ **(4)**. Applying Lemma 1.4.1 to (1) yields $C \Vdash \exists \sigma$, so the judgement (4) may be written $C, \Gamma \vdash \mathtt{t} : \sigma$. We have shown that (1) is derivable via the rules of Figure 1-8 and hm-Gen. In fact, it is possible to apply hm-Gen only once, at the end of the derivation.

1.5.1 Solution: Within the type system PCB(X), we have

$$\dfrac{\dfrac{\dfrac{}{\mathsf{z}_1 \preceq \mathsf{Z} \vdash \mathsf{z}_1 : \mathsf{Z}}\,\textsc{Var} \qquad \dfrac{}{\mathsf{z}_2 \preceq \mathsf{Y} \vdash \mathsf{z}_2 : \mathsf{Y}}\,\textsc{Var}}{\mathsf{let}\ \mathsf{z}_2 : \forall \mathsf{Z}[\mathsf{z}_1 \preceq \mathsf{Z}].\mathsf{Z}\ \mathsf{in}\ \mathsf{z}_2 \preceq \mathsf{Y} \vdash \mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2 : \mathtt{Y}}\,\textsc{Let}}{\mathsf{let}\ \mathsf{z}_1 : \mathtt{X}; \mathsf{z}_2 : \forall \mathsf{Z}[\mathsf{z}_1 \preceq \mathsf{Z}].\mathsf{Z}\ \mathsf{in}\ \mathsf{z}_2 \preceq \mathsf{Y} \vdash \lambda \mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2 : \mathtt{X} \rightarrow \mathtt{Y}}\,\textsc{Abs}$$

The type variable $\mathtt{Z}$, which occurs free in the left-hand instance of Var, is generalized. However, $\mathtt{z}_2$ does *not* receive the type scheme $\forall \mathtt{Z}.\mathtt{Z}$, which, as suggested earlier, is unsound; instead, it receives the *constrained* type scheme $\forall \mathsf{Z}[\mathsf{z}_1 \preceq \mathsf{Z}].\mathsf{Z}$. The latter is more restrictive than the former: indeed, the former claims that $\mathtt{z}_2$ has every type, while the latter only claims that every valid type for $\mathtt{z}_1$ is also a valid type for $\mathtt{z}_2$. Let us now examine the constraint $\mathsf{let}\ \mathsf{z}_1 : \mathtt{X}; \mathsf{z}_2 : \forall \mathsf{Z}[\mathsf{z}_1 \preceq \mathsf{Z}].\mathsf{Z}\ \mathsf{in}\ \mathsf{z}_2 \preceq \mathsf{Y}$, which appears at the root of the derivation. By C-InId and C-In*, it is equivalent to $\mathsf{let}\ \mathsf{z}_1 : \mathtt{X}\ \mathsf{in}\ \exists \mathsf{Z}.(\mathsf{z}_1 \preceq \mathsf{Z} \wedge \mathsf{Z} \leq \mathsf{Y})$ and to $\exists \mathsf{Z}.(\mathtt{X} \leq \mathsf{Z} \wedge \mathsf{Z} \leq \mathsf{Y})$, which by C-ExTrans is equivalent to $\mathtt{X} \leq \mathtt{Y}$. Thus, the judgement at the root of the above derivation may be written $\mathtt{X} \leq \mathtt{Y} \vdash \lambda \mathtt{z}_1.\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2 : \mathtt{X} \rightarrow \mathtt{Y}$. In other words, the expression $\mathtt{let}\ \mathtt{z}_2 = \mathtt{z}_1\ \mathtt{in}\ \mathtt{z}_2$ has type $\mathtt{X} \rightarrow \mathtt{Y}$ only under the assumption that $\mathtt{X}$ is a subtype of $\mathtt{Y}$, which is sound. Even though Let allows unrestricted generalization of type variables, it remains sound, because the type scheme that it produces typically has *free program identifiers*, such as $\forall \mathsf{Z}[\mathsf{z}_1 \preceq \mathsf{Z}].\mathsf{Z}$ above.

1.7.10 SOLUTION: Let $\mathcal{E} = \texttt{let z} = \mathcal{E}_1 \texttt{ in t}_1$ and $\mathcal{E}_1[\texttt{t}]/\mu \sqsubseteq \mathcal{E}_1[\texttt{t}']/\mu'$ **(1)**. Then,

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}[\texttt{t}]/\mu : \texttt{T}/M]\!] & \\
= & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ ((\mathsf{let}\ \texttt{z} : \forall\texttt{X}[[\![\mathcal{E}_1[\texttt{t}] : \texttt{X}]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!]) \wedge [\![\mu : M]\!]) & \textbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \texttt{z} : \forall\texttt{X}[[\![\mathcal{E}_1[\texttt{t}]/\mu : \texttt{X}/M]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & \textbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \texttt{z} : \forall\texttt{X}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ [\![\mathcal{E}_1[\texttt{t}]/\mu : \texttt{X}/M]\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & \textbf{(4)} \\
\Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M; \texttt{z} : \forall\texttt{X}\bar{\texttt{Y}}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}_1[\texttt{t}']/\mu' : \texttt{X}/M']\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & \textbf{(5)}
\end{array}$$

where (2) is by definition of constraint generation, where $\texttt{X} \notin ftv(\texttt{T}, M)$ **(6)**; (3) is by (6), C-LETAND, and by definition of constraint generation; (4) is by (6) and C-LETDUP; (5) follows from (1) and C-LETEX, for some $\bar{\texttt{Y}}$ and M' such that $\bar{\texttt{Y}} \# ftv(\texttt{X}, M)$ **(7)** and $ftv(M') \subseteq \bar{\texttt{Y}} \cup ftv(M)$ **(8)** and $dom(M') = dom(\mu')$ and M' extends M. Note that (6), (7) and (8) imply $\texttt{X} \notin ftv(M')$ **(9)**.

At this point, the type variables $\bar{\texttt{Y}}$, which determine the types of the newly allocated store cells, are universally quantified in the type scheme assigned to z, which is undesirable. We are stuck, because we cannot in general apply C-LETALL to hoist $\exists\bar{\texttt{Y}}$ out of the let constraint. Let us now assume that, by some external means, we are guaranteed $\bar{\texttt{Y}} = \varnothing$ **(10)**. Then, we may proceed as follows:

$$\begin{array}{lll}
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'; \texttt{z} : \forall\texttt{X}[\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}_1[\texttt{t}']/\mu' : \texttt{X}/M']\!]].\texttt{X}\ \mathsf{in}\ [\![\texttt{t}_1 : \texttt{T}]\!] & \textbf{(11)} \\
\equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\ M'\ \mathsf{in}\ [\![\mathcal{E}[\texttt{t}']/\mu' : \texttt{T}/M']\!] & \textbf{(12)}
\end{array}$$

where (11) follows from the fact the the memory locations that appear free in $[\![\texttt{t}_1 : \texttt{T}]\!]$ are members of $dom(\mu)$, thus are not members of $dom(M') \setminus dom(M)$; (12) is obtained by performing the steps that lead to (4) in reverse.

The requirement that $\bar{\texttt{Y}}$ be empty, that is, $ftv(M) = ftv(M')$, is classic (Tofte, 1988). How is it enforced? Assume that the left-hand side of every `let` construct is required to be a non-expansive expression. By assumptions (ii) and (iii), this invariant is preserved by reduction. So, $\mathcal{E}_1[\texttt{t}]$ must be non-expansive, which, by assumption (i), guarantees that the reduction step does not allocate new memory cells. Then, μ' is μ, so M' is M.

1.9.1 SOLUTION: We must first ensure that R-ADD respects $\sqsubseteq$ (Definition 1.7.5). Since the rule is pure, it is sufficient to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \hat{+} \hat{k}_2 : \texttt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \texttt{T}]\!]$. In fact, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\hat{k}_1 \hat{+} \hat{k}_2 : \texttt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\texttt{XY}.(\hat{+} \preceq \texttt{X} \to \texttt{Y} \to \texttt{T} \wedge \hat{k}_1 \preceq \texttt{X} \wedge \hat{k}_2 \preceq \texttt{Y}) & \textbf{(1)} \\
\equiv & \exists\texttt{XY}.(\texttt{int} \to \texttt{int} \to \texttt{int} \leq \texttt{X} \to \texttt{Y} \to \texttt{T} \wedge \texttt{int} \leq \texttt{X} \wedge \texttt{int} \leq \texttt{Y}) & \textbf{(2)} \\
\equiv & \exists\texttt{XY}.(\texttt{X} = \texttt{int} \wedge \texttt{Y} = \texttt{int} \wedge \texttt{int} \leq \texttt{T}) & \textbf{(3)} \\
\equiv & \texttt{int} \leq \texttt{T} & \textbf{(4)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\widehat{k_1 + k_2} : \texttt{T}]\!] & \textbf{(5)}
\end{array}$$

where (1) is by definition of constraint generation; (2) is by definition of Γ_0, by C-InId and C-In*; (3) is by C-Arrow and by antisymmetry of subtyping; (4) is by C-ExAnd and C-Name; (5) is again by definition of Γ_0, by C-InId and C-In*, and by definition of constraint generation.

Second, we must check that if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value.

We begin by checking that every value that is well-typed with type $\mathtt{int}$ is of the form $\hat{k}$. Indeed, suppose that $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{int}]\!]$ is satisfiable. Then, $\mathtt{v}$ cannot be a program variable, for a well-typed value must be closed. $\mathtt{v}$ cannot be a memory location m, for otherwise $\mathsf{ref}\, M(m) \leq \mathtt{int}$ would be satisfiable—but the type constructors ref and $\mathtt{int}$ are incompatible. $\mathtt{v}$ cannot be $\hat{+}$ or $\hat{+}\ \mathtt{v}'$, for otherwise $\mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \leq \mathtt{int}$ or $\mathtt{int} \rightarrow \mathtt{int} \leq \mathtt{int}$ would be satisfiable—but the type constructors $\rightarrow$ and $\mathtt{int}$ are incompatible. Similarly, $\mathtt{v}$ cannot be a λ-abstraction. Thus, $\mathtt{v}$ must be of the form $\hat{k}$, for it is the only case left.

Next, we note that, according to the constraint generation rules, if the configuration $\mathtt{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ (\mathtt{c} \preceq \mathtt{X}_1 \rightarrow \ldots \rightarrow \mathtt{X}_k \rightarrow \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge \ldots \wedge [\![\mathtt{v}_k : \mathtt{X}_k]\!])$ is satisfiable. We now reason by cases on $\mathtt{c}$.

∘ *Case* $\mathtt{c}$ is $\hat{k}$. Then, $\Gamma_0(\mathtt{c})$ is $\mathtt{int}$. Because the type constructors $\mathtt{int}$ and $\rightarrow$ are incompatible with each other, this implies $k = 0$. Since $\hat{k}$ is a constructor, the expression is a value.

∘ *Case* $\mathtt{c}$ is $\hat{+}$. We may assume $k \geq 2$, because otherwise the expression is a value. Then, $\Gamma_0(\mathtt{c})$ is $\mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int}$, so, by C-Arrow, the above constraint entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ (\mathtt{X}_1 \leq \mathtt{int} \wedge \mathtt{X}_2 \leq \mathtt{int} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}_2]\!])$, which, by Lemma 1.6.3, entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ ([\![\mathtt{v}_1 : \mathtt{int}]\!] \wedge [\![\mathtt{v}_2 : \mathtt{int}]\!])$. Thus, $\mathtt{v}_1$ and $\mathtt{v}_2$ are well-typed with type $\mathtt{int}$. By the remark above, they must be integer literals $\hat{k}_1$ and $\hat{k}_2$. As a result, the configuration is reducible by R-Add.

1.9.5 Solution: We must first ensure that R-Ref, R-Deref and R-Assign respect $\sqsubseteq$ (Definition 1.7.5).

∘ *Case* R-Ref. The reduction is $\mathtt{ref}\ \mathtt{v}/\varnothing \longrightarrow m/(m \mapsto \mathtt{v})$, where $m \notin \mathit{fpi}(\mathtt{v})$ **(1)**. Let $\mathtt{T}$ be an arbitrary type. According to Definition 1.7.5, the goal is to show that there exist a set of type variables $\bar{\mathtt{Y}}$ and a store type M' such that $\bar{\mathtt{Y}}\ \#\ \mathit{ftv}(\mathtt{T})$ and $\mathit{ftv}(M') \subseteq \bar{\mathtt{Y}}$ and $\mathit{dom}(M') = \{m\}$ and $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{ref}\ \mathtt{v} : \mathtt{T}]\!]$

entails $\exists\bar{\mathtt{Y}}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![m/(m \mapsto \mathtt{v}) : \mathtt{T}/M']\!]$. Now, we have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{ref}\ \mathtt{v} : \mathtt{T}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{XY}.(\mathtt{Y} \rightarrow \mathsf{ref}\,\mathtt{Y} \leq \mathtt{X} \rightarrow \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{X}]\!]) & (\mathbf{2}) \\ \equiv & \exists\mathtt{Y}.\mathsf{let}\ \Gamma_0\ \mathsf{in}\ (\mathsf{ref}\,\mathtt{Y} \leq \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & (\mathbf{3}) \\ \equiv & \exists\mathtt{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ (m \preceq \mathtt{T} \wedge [\![\mathtt{v} : M'(m)]\!]) & (\mathbf{4}) \\ \equiv & \exists\mathtt{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\, M'\ \mathsf{in}\ [\![m/(m \mapsto \mathtt{v}) : \mathtt{T}/M']\!] & (\mathbf{5}) \end{array}$$

where (2) is by definition of constraint generation and by definition of $\Gamma_0(\mathtt{ref})$; (3) is by C-Arrow, Lemma 1.6.4, and C-InEx; (4) assumes M' is defined as $m \mapsto \mathtt{Y}$, and follows from (1), C-InId and C-In*; and (5) is by definition of constraint generation.

Subcase R-Deref. The reduction is $!m/(m \mapsto \mathtt{v}) \longrightarrow \mathtt{v}/(m \mapsto \mathtt{v})$. Let $\mathtt{T}$ be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![!m/(m \mapsto \mathtt{v}) : \mathtt{T}/M]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{XY}.(\mathsf{ref}\,\mathtt{Y} \rightarrow \mathtt{Y} \leq \mathtt{X} \rightarrow \mathtt{T} \wedge m \preceq \mathtt{X} \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{1}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{XY}.(\mathsf{ref}\, M(m) \leq \mathtt{X} \leq \mathsf{ref}\,\mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{T} \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{2}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{Y}.(M(m) = \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{T} \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{3}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ (M(m) \leq \mathtt{T} \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{4}) \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ ([\![\mathtt{v} : \mathtt{T}]\!] \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{5}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{v}/(m \mapsto \mathtt{v}) : \mathtt{T}/M]\!] & (\mathbf{6}) \end{array}$$

where (1) is by definition of constraint generation and by definition of $\Gamma_0(!)$; (2) is by C-Arrow and C-InId; (3) follows from C-ExTrans and from the fact that ref is an invariant type constructor; (4) is by C-NameEq; (5) is by Lemma 1.6.3 and C-Dup; and (6) is again by definition of constraint generation.

∘ *Case* R-Assign. The reduction is $m := \mathtt{v}/(m \mapsto \mathtt{v}_0) \longrightarrow \mathtt{v}/(m \mapsto \mathtt{v})$. Let $\mathtt{T}$ be an arbitrary type and let M be a store type of domain $\{m\}$. We have

$$\begin{array}{lll} & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![m := \mathtt{v}/(m \mapsto \mathtt{v}_0) : \mathtt{T}/M]\!] & \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![m := \mathtt{v} : \mathtt{T}]\!] & (\mathbf{1}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{XYZ}.(\mathsf{ref}\,\mathtt{Z} \rightarrow \mathtt{Z} \rightarrow \mathtt{Z} \leq \mathtt{X} \rightarrow \mathtt{Y} \rightarrow \mathtt{T} \wedge m \preceq \mathtt{X} \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & (\mathbf{2}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{XYZ}.(\mathsf{ref}\, M(m) \leq \mathtt{X} \leq \mathsf{ref}\,\mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Y}]\!] \wedge \mathtt{Y} \leq \mathtt{Z}) & (\mathbf{3}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ \exists\mathtt{Z}.(M(m) = \mathtt{Z} \wedge \mathtt{Z} \leq \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Z}]\!]) & (\mathbf{4}) \\ \equiv & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ (M(m) \leq \mathtt{T} \wedge [\![\mathtt{v} : M(m)]\!]) & (\mathbf{5}) \\ \Vdash & \mathsf{let}\ \Gamma_0; \mathsf{ref}\, M\ \mathsf{in}\ [\![\mathtt{v}/(m \mapsto \mathtt{v}) : \mathtt{T}/M]\!] & (\mathbf{6}) \end{array}$$

where (1) and (2) are by definition of constraint generation; (3) is by C-Arrow and C-InId; (4) is by C-ExTrans, Lemma 1.6.4, and from the fact that ref is an invariant type constructor; (5) is by C-NameEq; and (6) is obtained as in the previous case.

Second, we must check that if the configuration $\mathtt{c}\ \mathtt{v}_1\ \dots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{c}\ \mathtt{v}_1\ \dots\ \mathtt{v}_k$ is a value. We only give a sketch of this proof; see the solution to Exercise 1.9.1 for details of a similar proof.

We begin by checking that every value that is well-typed with a type of the form $\mathsf{ref}\,\mathtt{T}$ is a memory location. This assertion relies on the fact that the type constructor ref is isolated.

Next, we note that, according to the constraint generation rules, if the configuration $\mathtt{c}\ \mathtt{v}_1\ \dots\ \mathtt{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\,M\ \mathsf{in}\ (\mathtt{c} \preceq \mathtt{X}_1 \to \dots \to \mathtt{X}_k \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge \dots \wedge [\![\mathtt{v}_k : \mathtt{X}_k]\!])$ is satisfiable. We now reason by cases on $\mathtt{c}$.

∘ *Case* $\mathtt{c}$ is $\mathtt{ref}$. If $k = 0$, then the expression is a value; otherwise, it is reducible by R-Ref.

∘ *Case* $\mathtt{c}$ is $!$. We may assume $k \geq 1$, because otherwise the expression is a value. Then, by definition of $\Gamma_0(!)$, the above constraint entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\,M\ \mathsf{in}\ \exists \mathtt{Y}.(\mathsf{ref}\,\mathtt{Y} \to \mathtt{Y} \leq \mathtt{X}_1 \to \dots \to \mathtt{X}_k \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!])$, which, by C-Arrow, Lemma 1.6.3, and C-InEx, entails $\exists \mathtt{Y}.\mathsf{let}\ \Gamma_0; \mathsf{ref}\,M\ \mathsf{in}\ [\![\mathtt{v}_1 : \mathsf{ref}\,\mathtt{Y}]\!]$. Thus, $\mathtt{v}_1$ is well-typed with a type of the form $\mathsf{ref}\,\mathtt{Y}$. By the remark above, $\mathtt{v}_1$ must be a memory location m. Furthermore, because every well-typed configuration is closed, m must be a member of $dom(\mu)$. As a result, the configuration $\mathtt{ref}\ \mathtt{v}_1\ \dots\ \mathtt{v}_k/\mu$ is reducible by R-Deref.

∘ *Case* $\mathtt{c}$ is $:=$. We may assume $k \geq 2$, because otherwise the expression is a value. As above, we check that $\mathtt{v}_1$ must be a memory location and a member of $dom(\mu)$. Thus, the configuration is reducible by R-Assign.

1.9.6 Solution: We must first ensure that R-Fix respects $\sqsubseteq$ (Definition 1.7.5). Since the rule is pure, it is sufficient to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{fix}\ \mathtt{v}_1\ \mathtt{v}_2 : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v}_1\ (\mathtt{fix}\ \mathtt{v}_1)\ \mathtt{v}_2 : \mathtt{T}]\!]$. Let C stand for the constraint $\mathtt{fix} \preceq ((\mathtt{X} \to \mathtt{Y}) \to (\mathtt{X} \to \mathtt{Y})) \to \mathtt{X} \to \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{T} \wedge [\![\mathtt{v}_1 : (\mathtt{X} \to \mathtt{Y}) \to (\mathtt{X} \to \mathtt{Y})]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}]\!]$. We have

$$\begin{array}{cll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{fix}\ \mathtt{v}_1\ \mathtt{v}_2 : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}_1\mathtt{X}_2.(\mathtt{fix} \preceq \mathtt{X}_1 \to \mathtt{X}_2 \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}_2]\!]) & \mathbf{(1)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}_1\mathtt{X}_2\mathtt{X}\mathtt{Y}.(((\mathtt{X} \to \mathtt{Y}) \to (\mathtt{X} \to \mathtt{Y})) \to \mathtt{X} \to \mathtt{Y} \leq \mathtt{X}_1 \to \mathtt{X}_2 \to \mathtt{T} & \\
 & \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}_2]\!]) & \mathbf{(2)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{Y}.(\mathtt{Y} \leq \mathtt{T} \wedge [\![\mathtt{v}_1 : (\mathtt{X} \to \mathtt{Y}) \to (\mathtt{X} \to \mathtt{Y})]\!] \wedge [\![\mathtt{v}_2 : \mathtt{X}]\!]) & \mathbf{(3)} \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\mathtt{Y}.C & \mathbf{(4)}
\end{array}$$

where (1) is by definition of constraint generation; (2) is by definition of $\Gamma_0(\mathtt{fix})$; (3) is by C-Arrow and Lemma 1.6.4; (4) is by definition of $\Gamma_0(\mathtt{fix})$. By Theorem 1.6.2 and Weaken, the judgements $C \vdash \mathtt{v}_1 : (\mathtt{X} \to \mathtt{Y}) \to$

$(\mathtt{X} \rightarrow \mathtt{Y})$ and $C \vdash \mathtt{v}_2 : \mathtt{X}$ hold. By VAR, WEAKEN, APP, and SUB, it follows that $C \vdash \mathtt{v}_1\,(\mathtt{fix}\,\mathtt{v}_1)\,\mathtt{v}_2 : \mathtt{T}$ holds. By Theorem 1.6.6, this implies $C \Vdash [\![\mathtt{v}_1\ (\mathtt{fix}\ \mathtt{v}_1)\ \mathtt{v}_2 : \mathtt{T}]\!]$. By congruence of entailment and by C-EX*, (4) entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v}_1\ (\mathtt{fix}\ \mathtt{v}_1)\ \mathtt{v}_2 : \mathtt{T}]\!]$.

Second, we must check that if the configuration $\mathtt{fix}\,\mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ (where $k \geq 0$) is well-typed, then either it is reducible, or $\mathtt{fix}\,\mathtt{v}_1\ \ldots\ \mathtt{v}_k$ is a value. This is immediate, for it is a value when $k < 2$, and it is reducible by R-FIX when $k \geq 2$.

We now recall that the construct $\mathtt{letrec}\ \mathtt{f} = \lambda\mathtt{z}.\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ provided by ML-the-programming-language may be viewed as syntactic sugar for $\mathtt{let}\ \mathtt{f} = \mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1)\ \mathtt{in}\ \mathtt{t}_2$, and set forth to discover the constraint generation rule that arises out of such a definition. We have

$$\begin{array}{rll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1) : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{Z}.(\mathtt{fix} \preceq \mathtt{Z} \rightarrow \mathtt{T} \wedge [\![\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1 : \mathtt{Z}]\!]) & (\mathbf{1}) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{XY}.(\mathtt{X} \rightarrow \mathtt{Y} \leq \mathtt{T} \wedge [\![\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1 : (\mathtt{X} \rightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \rightarrow \mathtt{Y})]\!]) & (\mathbf{2}) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists\mathtt{XY}.(\mathtt{X} \rightarrow \mathtt{Y} \leq \mathtt{T} \wedge \mathsf{let}\ \mathtt{f} : \mathtt{X} \rightarrow \mathtt{Y}; \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{Y}]\!]) & (\mathbf{3})
\end{array}$$

where (1) is by definition of constraint generation; (2) is by definition of $\Gamma_0(\mathtt{fix})$, by C-ARROW, and by Lemma 1.6.4; and (3) follows from Lemma 1.6.5. This allows us to write

$$\begin{array}{rll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{let}\ \mathtt{f} = \mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1)\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0; \mathtt{f} : \forall\mathtt{Z}[[\![\mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{z}.\mathtt{t}_1) : \mathtt{Z}]\!]].\mathtt{Z}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{4}) \\
\equiv & \mathsf{let}\ \Gamma_0; \mathtt{f} : \forall\mathtt{Z}[\exists\mathtt{XY}.(\mathtt{X} \rightarrow \mathtt{Y} \leq \mathtt{Z} \wedge \mathsf{let}\ \mathtt{f} : \mathtt{X} \rightarrow \mathtt{Y}; \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{Y}]\!])].\mathtt{Z}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{5}) \\
\equiv & \mathsf{let}\ \Gamma_0; \mathtt{f} : \forall\mathtt{XY}[\mathsf{let}\ \mathtt{f} : \mathtt{X} \rightarrow \mathtt{Y}; \mathtt{z} : \mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_1 : \mathtt{Y}]\!]].\mathtt{X} \rightarrow \mathtt{Y}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{6})
\end{array}$$

where (4) is by definition of constraint generation; (5) follows from C-LETDUP and from the previous series of equivalences; (6) is by C-LETEX, C-EXTRANS and Lemma 1.3.22.

1.9.21 SOLUTION: We have

$$\begin{array}{rll}
& [\![\mathtt{match}\ \mathtt{t}_1\ \mathtt{with}\ \mathtt{z}\,.\,\mathtt{t}_2 : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \forall\mathtt{XX}'[[\![\mathtt{t}_1 : \mathtt{X}]\!] \wedge \mathsf{let}\ \mathtt{z} : \mathtt{X}'\ \mathsf{in}\ [\![\mathtt{X} : \mathtt{z}]\!]].(\mathtt{z} : \mathtt{X}')\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{1}) \\
\equiv & \mathsf{let}\ \mathtt{z} : \forall\mathtt{X}'[\exists\mathtt{X}.([\![\mathtt{t}_1 : \mathtt{X}]\!] \wedge \mathtt{X} \leq \mathtt{X}')].\mathtt{X}'\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{2}) \\
\equiv & \mathsf{let}\ \mathtt{z} : \forall\mathtt{X}'[[\![\mathtt{t}_1 : \mathtt{X}']\!]].\mathtt{X}'\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{3}) \\
\equiv & [\![\mathtt{let}\ \mathtt{z} = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{4})
\end{array}$$

where (1) is by definition of constraint generation for `match`; (2) is by definition of constraint generation for patterns, by C-INID, C-IN*, and C-LETEX; (3) is by Lemma 1.6.4; (4) is by definition of constraint generation for `let`.

1.9.26 SOLUTION: The type scheme $\forall\bar{\mathtt{X}}.\mathtt{T} \rightarrow \mathtt{T}$ may be written $\forall\bar{\mathtt{X}}.[\mathtt{X} \mapsto \mathtt{T}](\mathtt{X} \rightarrow \mathtt{X})$. Furthermore, $\bar{\mathtt{X}}\ \#\ \forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$ holds. Thus, $\forall\bar{\mathtt{X}}.\mathtt{T} \rightarrow \mathtt{T}$ is an instance of $\forall\mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$

in the sense of DM-INST'. Since DM-INST' is an admissible rule for the type system DM, and since it is clear that the identity function $\lambda z.z$ has type $\forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}$, it must also have type $\forall \bar{\mathtt{X}}.\mathtt{T} \rightarrow \mathtt{T}$. (A more direct proof of this fact would not be difficult.) So, the destructor $(\cdot : \exists \bar{\mathtt{X}}.\mathtt{T})$ has not only identity semantics, but also an identity type. This shows that our definitions are sound.

Let us now check requirement (i) of Definition 1.7.6. Since R-ANNOTATION is pure, it suffices to show that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\mathtt{v} : \exists \bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}']\!]$. Now, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![(\mathtt{v} : \exists \bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\bar{\mathtt{X}}.(\mathtt{T} \rightarrow \mathtt{T} \leq \mathtt{X} \rightarrow \mathtt{T}' \wedge [\![\mathtt{v} : \mathtt{X}]\!]) & (1) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}\bar{\mathtt{X}}.(\mathtt{X} \leq \mathtt{T} \leq \mathtt{T}' \wedge [\![\mathtt{v} : \mathtt{X}]\!]) & (2) \\
\Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}']\!] & (3)
\end{array}$$

where (1) is by definition of constraint generation and by definition of $\Gamma_0((\cdot : \exists \bar{\mathtt{X}}.\mathtt{T}))$; (2) is by C-ARROW; and (3) follows from Lemma 1.6.3 and C-EX*.

1.10.5 SOLUTION: We have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}.[\![(\lambda z.z \mathbin{\hat{+}} \hat{1} : \forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}.(\forall \mathtt{X}.[\![\lambda z.z \mathbin{\hat{+}} \hat{1} : \mathtt{X} \rightarrow \mathtt{X}]\!] \wedge \exists \mathtt{X}.(\mathtt{X} \rightarrow \mathtt{X} \leq \mathtt{Z})) & (\mathbf{1}) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \forall \mathtt{X}.\mathsf{let}\ z : \mathtt{X}\ \mathsf{in}\ [\![z \mathbin{\hat{+}} \hat{1} : \mathtt{X}]\!] & (\mathbf{2}) \\
\equiv & \forall \mathtt{X}.(\mathtt{int} \rightarrow \mathtt{int} \rightarrow \mathtt{int} \leq \mathtt{X} \rightarrow \mathtt{int} \rightarrow \mathtt{X}) & (\mathbf{3}) \\
\equiv & \forall \mathtt{X}.(\mathtt{X} = \mathtt{int}) & (\mathbf{4}) \\
\equiv & \mathsf{false} & (\mathbf{5})
\end{array}$$

where (1) is by definition of constraint generation for universal type annotations; (2) is obtained by restricting the scope of $\exists \mathtt{Z}$ to the second conjunct, then dropping the latter altogether, since it is equivalent to true, and by Lemma 1.6.5; (3) is obtained by definition of constraint generation, by definition of $\Gamma_0(\hat{+})$ and of $\Gamma_0(\hat{1})$, and by a few simple equivalence laws; (4) follows from C-ARROW and antisymmetry of subtyping; (5) follows from the fact that $\mathtt{int}$ and (say) $\mathtt{int} \rightarrow \mathtt{int}$ have distinct interpretations, since the type constructors $\mathtt{int}$ and $\rightarrow$ are incompatible. On the other hand, we have

$$\begin{array}{lll}
 & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Z}.[\![(\lambda z.z : \forall \mathtt{X}.\mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \forall \mathtt{X}.\mathsf{let}\ z : \mathtt{X}\ \mathsf{in}\ [\![z : \mathtt{X}]\!] & (\mathbf{1}) \\
\equiv & \forall \mathtt{X}.(\mathtt{X} \leq \mathtt{X}) & (\mathbf{2}) \\
\equiv & \mathsf{true} & (\mathbf{3})
\end{array}$$

where (1) is obtained as above; (2) by definition of constraint generation, C-INID and C-IN*; (3) is by reflexivity of subtyping.

1.10.6 SOLUTION: Under the naïve constraint generation rule for universal type variable introduction, the constraint $[\![\forall \mathtt{X}.(\lambda \mathsf{z}.\mathsf{z} : \mathtt{X} \rightarrow \mathtt{X}) : \mathtt{Z}]\!]$ is equivalent to $\forall \mathtt{X}.([\![\lambda \mathsf{z}.\mathsf{z} : \mathtt{X} \rightarrow \mathtt{X}]\!] \wedge \mathtt{X} \rightarrow \mathtt{X} \leq \mathtt{Z})$. Since the first conjunct is a tautology, this is in turn equivalent to $\forall \mathtt{X}.(\mathtt{X} \rightarrow \mathtt{X} \leq \mathtt{Z})$. In a nondegenerate free term model where subtyping is interpreted as equality, this constraint is unsatisfiable. In a non-structural subtyping model equipped with a least type $\bot$ and a greatest type $\top$, it is equivalent to $\bot \rightarrow \top \leq \mathtt{Z}$. This is a pretty restrictive constraint: since no value has type $\bot$, a function whose type is (a supertype of) $\bot \rightarrow \top$ cannot ever be invoked at runtime. This situation is clearly unsatisfactory. Checking that $\forall \mathtt{X}.[\![\lambda \mathsf{z}.\mathsf{z} : \mathtt{X} \rightarrow \mathtt{X}]\!]$ holds was indeed part of our intent, but constraining $\mathtt{Z}$ to be a supertype of $\mathtt{X} \rightarrow \mathtt{X}$ for *every* $\mathtt{X}$ was not.

1.10.7 SOLUTION: Let $\bar{\mathtt{X}} \supseteq ftv(\mathtt{T})$ **(1)** and $\bar{\mathtt{X}} \# ftv(\mathtt{t})$ **(2)**. We may assume, *w.l.o.g.*, $\bar{\mathtt{X}} \# ftv(\mathtt{T}')$ **(3)**. By (1), (2), (3), and by definition of constraint generation for local universal type annotations, $[\![(\mathtt{t} : \forall \bar{\mathtt{X}}.\mathtt{T}) : \mathtt{T}']\!]$ is well-defined and is $\forall \bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \wedge \exists \bar{\mathtt{X}}.(\mathtt{T} \leq \mathtt{T}')$ **(4)**. By (3) and by definition of constraint generation for introduction of universal type variables and for general type annotations, $[\![\forall \bar{\mathtt{X}}.(\mathtt{t} : \mathtt{T}) : \mathtt{T}']\!]$ is $\forall \bar{\mathtt{X}}.\exists \mathtt{Z}.([\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{Z}) \wedge \exists \bar{\mathtt{X}}.([\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}')$, where $\mathtt{Z}$ is fresh, which we may immediately simplify to $\forall \bar{\mathtt{X}}.[\![\mathtt{t} : \mathtt{T}]\!] \wedge \exists \bar{\mathtt{X}}.([\![\mathtt{t} : \mathtt{T}]\!] \wedge \mathtt{T} \leq \mathtt{T}')$ **(5)**. Using C-EXAND and Lemma 1.10.1, it is straightforward to check that (4) and (5) are equivalent.

1.10.9 SOLUTION: We have

$$\begin{array}{lll} & \exists \mathtt{Z}.[\![\lambda \mathsf{z}.\forall \mathtt{X}.(\mathsf{z} : \mathtt{X}) : \mathtt{Z}]\!] & \\ \Vdash & \exists \mathtt{Z}_1 \mathtt{Z}_2.\mathsf{let}\ \mathsf{z} : \mathtt{Z}_1\ \mathsf{in}\ [\![\forall \mathtt{X}.(\mathsf{z} : \mathtt{X}) : \mathtt{Z}_2]\!] & (\mathbf{1}) \\ \Vdash & \exists \mathtt{Z}_1.\forall \mathtt{X}.(\mathtt{Z}_1 \leq \mathtt{X}) & (\mathbf{2}) \end{array}$$

where (1) is by definition of constraint generation for λ-abstractions, dropping the constraint that relates $\mathtt{Z}$, $\mathtt{Z}_1$, and $\mathtt{Z}_2$; (2) is by definition of constraint generation for universal type variable introduction, this time dropping information about $\mathtt{Z}_2$. Now, in a nondegenerate equality model, the constraint (2) is equivalent to false. In fact, for (2) to be satisfiable, the interpretation of subtyping must admit a least element $\bot$. We now see that $[\![\lambda \mathsf{z}.\forall \mathtt{X}.(\mathsf{z} : \mathtt{X}) : \mathtt{Z}]\!]$ is a very restrictive constraint. Indeed, it requires z to have every type at once. Because z is λ-bound—hence monomorphic—it must in fact have type $\bot$. On the other hand, we have

$$\begin{array}{lll} & \exists \mathtt{Z}.[\![\forall \mathtt{X}.\lambda \mathsf{z}.(\mathsf{z} : \mathtt{X}) : \mathtt{Z}]\!] & \\ \equiv & \forall \mathtt{X}.\exists \mathtt{Z}.[\![\lambda \mathsf{z}.(\mathsf{z} : \mathtt{X}) : \mathtt{Z}]\!] & (\mathbf{1}) \\ \equiv & \forall \mathtt{X}.\exists \mathtt{Z}\mathtt{Z}_1\mathtt{Z}_2.(\mathtt{Z}_1 \leq \mathtt{X} \wedge \mathtt{X} \leq \mathtt{Z}_2 \wedge \mathtt{Z}_1 \rightarrow \mathtt{Z}_2 \leq \mathtt{Z}) & (\mathbf{2}) \\ \equiv & \mathsf{true} & (\mathbf{3}) \end{array}$$

where (1) is by definition of constraint generation for universal type variable introduction, dropping the second conjunct, which is entailed by the first; (2) is by Lemma 1.6.5, by definition of constraint generation for general type annotations, and by a few simple equivalence laws; (3) follows from C-NameEq and the witness substitution $[\mathtt{Z}_1 \mapsto \mathtt{X}, \mathtt{Z}_2 \mapsto \mathtt{X}, \mathtt{Z} \mapsto (\mathtt{X} \to \mathtt{X})]$.

1.10.10 Solution: We have

$$\begin{array}{rll} & [\![\mathtt{letrec}\,\mathtt{f} : \mathtt{S} = \lambda \mathtt{z}.\mathtt{t}_1 \ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}]\!] & \\ \equiv & \mathsf{let}\ \mathtt{f} : \forall \mathtt{X}[[\![\mathtt{fix}\ \mathtt{f} : \mathtt{S}.\lambda \mathtt{z}.\mathtt{t}_1 : \mathtt{X}]\!]].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{1}) \\ \equiv & \mathsf{let}\ \mathtt{f} : \forall \mathtt{X}[\mathsf{let}\ \mathtt{f} : \mathtt{S}\ \mathsf{in}\ [\![\lambda \mathtt{z}.\mathtt{t}_1 : \mathtt{S}]\!] \wedge \mathtt{S} \preceq \mathtt{X}].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{2}) \\ \equiv & \mathsf{let}\ \mathtt{f} : \mathtt{S}\ \mathsf{in}\ [\![\lambda \mathtt{z}.\mathtt{t}_1 : \mathtt{S}]\!] \wedge \mathsf{let}\ \mathtt{f} : \forall \mathtt{X}[\mathtt{S} \preceq \mathtt{X}].\mathtt{X}\ \mathsf{in}\ [\![\mathtt{t}_2 : \mathtt{T}]\!] & (\mathbf{3}) \\ \equiv & \mathsf{let}\ \mathtt{f} : \mathtt{S}\ \mathsf{in}\ ([\![\lambda \mathtt{z}.\mathtt{t}_1 : \mathtt{S}]\!] \wedge [\![\mathtt{t}_2 : \mathtt{T}]\!]) & (\mathbf{4}) \end{array}$$

where (1) is by definition of the `letrec` syntactic sugar and by the definition of constraint generation for `let` constructs; we have $\mathtt{X} \notin \mathit{ftv}(\mathtt{S}, \mathtt{t}_1)$; (2) is by definition of constraint generation for `fix`; (3) is by C-LetAnd; (4) follows from the equivalence between the type schemes $\forall \mathtt{X}[\mathtt{S} \preceq \mathtt{X}].\mathtt{X}$ and $\mathtt{S}$—which itself is a direct consequence of C-ExTrans—and from C-InAnd.

1.11.16 Solution: We reason simultaneously in both the subtyping model or the equal-only model, that is, we only rely on properties that are valid in both models.

We must first ensure that rules rd-Default, rd-Found, and rd-Follow respect (Definition 1.7.5).

∘ *Case* rd-Default. The reduction is $\{\mathtt{v}\}.\{\ell\} \xrightarrow{\delta} \mathtt{v}$, which is pure. Therefore, it is sufficient to establish that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{v}\}.\{\ell\} : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!]$. In fact, we have:

$$\begin{array}{rll} & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{v}\}.\{\ell\} : \mathtt{T}]\!] & \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{XY}.(\cdot.\{\ell\} \preceq \mathtt{X} \to \mathtt{T} \wedge \{\cdot\} \preceq \mathtt{Y} \to \mathtt{X} \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & (\mathbf{1}) \\ \equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{XY}.(\exists \mathtt{X}_1\mathtt{X}_2.(\Pi(\ell : \mathtt{X}_1\ ;\ \mathtt{X}_2) \to \mathtt{X}_1 \leq \mathtt{X} \to \mathtt{T}) & (\mathbf{2}) \\ & \qquad \wedge \exists \mathtt{Y}_1.(\mathtt{Y}_1 \to \Pi(\partial \mathtt{Y}_1) \leq \mathtt{Y} \to \mathtt{X}) \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & \\ \Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{X}_2\mathtt{Y}.(\partial \mathtt{Y} \leq (\ell : \mathtt{X}_1\ ;\ \mathtt{X}_2) \wedge \mathtt{X}_1 \leq \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & (\mathbf{3}) \\ \Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Y}.(\mathtt{Y} \leq \mathtt{X}_1 \wedge \mathtt{X}_1 \leq \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Y}]\!]) & (\mathbf{4}) \\ \Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!] & (\mathbf{5}) \end{array}$$

where (1) is by definition of constraint generation; (2) is by definition of Γ_0, C-InId; (3) by variances of Π, ℓ, and $\to$, C-And, C-Ex*, C-ExAnd; (4) by C-Row-DL and covariance of ℓ; (5) by Lemma 1.6.3.

∘ *Case* rd-Found: The reduction is $\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}.\{\ell\} \xrightarrow{\delta} \mathtt{v}$. It suffices to establish $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}.\{\ell\} : \mathtt{T}]\!]$ entails $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!]$. In fact, we have:

$$\begin{array}{lll}
& \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}.\{\ell\} : \mathtt{T}]\!] & \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{XYY'}.(\cdot.\{\ell\} \preceq \mathtt{X} \to \mathtt{T} \wedge \{\cdot\ \mathsf{with}\ \ell = \cdot\} \preceq \mathtt{Y} \to \mathtt{Y'} \to \mathtt{X} \wedge & \\
& \qquad \wedge [\![\mathtt{w} : \mathtt{Y}]\!] \wedge [\![\mathtt{v} : \mathtt{Y'}]\!]) & (\mathbf{1}) \\
\equiv & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{XYY'}.(\exists \mathtt{X}_1\mathtt{X}_2.(\Pi(\ell : \mathtt{X}_1\ ;\ \mathtt{X}_2) \to \mathtt{X}_1 \le \mathtt{X} \to \mathtt{T}) & \\
& \quad \wedge \exists \mathtt{Y}_1\mathtt{Y}_2\mathtt{Y}_3.(\Pi(\ell : \mathtt{Y}_1\ ;\ \mathtt{Y}_3) \to \mathtt{Y}_2 \to \Pi(\ell : \mathtt{Y}_2\ ;\ \mathtt{Y}_3) \le \mathtt{Y} \to \mathtt{Y'} \to \mathtt{X}) & \\
& \qquad \wedge [\![\mathtt{w} : \mathtt{Y}]\!] \wedge [\![\mathtt{v} : \mathtt{Y'}]\!]) & (\mathbf{2}) \\
\Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ \exists \mathtt{Y'X}_1\mathtt{Y}_2.(\mathtt{Y'} \le \mathtt{Y}_2 \wedge \mathtt{Y}_2 \le \mathtt{X}_1 \wedge \mathtt{X}_1 \le \mathtt{T} \wedge [\![\mathtt{v} : \mathtt{Y'}]\!]) & (\mathbf{3}) \\
\Vdash & \mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \mathtt{T}]\!] & (\mathbf{4})
\end{array}$$

where (1) is by definition of constraint generation; (2) is by definition of Γ_0, C-InId; (3) by variances of Π, ℓ, and $\to$, C-And, C-Ex*, C-ExAnd; (4) by Lemma 1.6.3.

∘ *Case* RD-Follow The proof is similar to the previous case.

We must now check that if the configuration $F\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ is is well-typed, then either it is reducible, or it is a value.

We begin by checking that every value that is well-typed with type $\Pi\ \mathtt{T}$ is a record value, that is, either of the form $\{\mathtt{v'}\}$ or $\{\mathtt{v''}\ \mathsf{with}\ \ell' = \mathtt{v'}\}$. Indeed, suppose that $\mathsf{let}\ \Gamma_0\ \mathsf{in}\ [\![\mathtt{v} : \Pi\ \mathtt{T}]\!]$ is satisfiable. Then, v cannot be a program variable, for a well-typed value must be closed; v cannot be a memory location m, for otherwise $\mathsf{ref}\ M(m) \le \Pi\ \mathtt{T}$ would be satisfiable—but the top type constructors ref and Π are incompatible (since Π is isolated); $\mathtt{v}$ cannot be a partial application of a constructor or a primitive, nor a λ-abstraction, since otherwise $\mathtt{T'} \to \mathtt{T''} \le \Pi\ \mathtt{T}$ would be satisfiable but the top type constructors $\to$ and Π are incompatible (since they are both isolated); thus $\mathtt{v}$ must either be of the form $\{\mathtt{v}\}$ or $\{\mathtt{w}\ \mathsf{with}\ \ell = \mathtt{v}\}$, for these are the only left cases.

Next, we note that, according to the constraint generation rules, if the configuration $\mathsf{c}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\mathsf{c} \preceq \mathtt{X}_1 \to \ldots \to \mathtt{X}_k \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge \ldots \wedge [\![\mathtt{v}_k : \mathtt{X}_k]\!])$ is satisfiable. We now reason by cases on c.

∘ *Case* c is $\{\cdot\}$. We may asume $k \ge 2$, since otherwise, the expression is a value. Then $\Gamma_0(\mathsf{c})$ is $\forall \mathtt{XY}.\mathtt{X} \to \Pi(\partial \mathtt{X})$, so by C-InId and C-arrow the above constraint entails $\exists \mathtt{X}.(\Pi(\partial \mathtt{X}) \le \mathtt{X}_2 \to \ldots \to \mathtt{T})$, which by C-Class-I entails $\mathtt{false}$ since $\to$ and Π are imcompatible. Thus, this case cannot occur.

∘ *Case* c is $\{\cdot\ \mathsf{with}\ \ell = \cdot\}$. Similar to the previous case.

∘ *Case* c is $\cdot.\{\ell\}$. We may asume $k \ge 1$, since otherwise, the expression is a value. Then $\Gamma_0(\mathsf{c})$ is $\forall \mathtt{XY}.\Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y}) \to \mathtt{X}$, so by C-InId and C-arrow the above constraint entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ (\exists \mathtt{XY}.(\mathtt{X}_1 \le \Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y})) \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!])$, which by lemma 1.6.3 entails $\mathsf{let}\ \Gamma_0; \mathsf{ref}\ M\ \mathsf{in}\ \exists \mathtt{XY}.[\![\mathtt{v}_1 : \Pi(\ell : \mathtt{X}\ ;\ \mathtt{Y})]\!]$. Thus $\mathtt{v}_1$ is a record value, that is, either of the form $\{\mathtt{v'}\}$ and the configuration is reducible to $\mathtt{v}'$ or of the form $\{\mathtt{v}'' \text{ with } \ell' = \mathtt{v}'\}$ and the configuration is reducible to either $\mathtt{v}'$ or $\mathtt{v}''.\{\ell\}$.

1.11.17 SOLUTION: We add a collection of destructors $\cdot[\ell_1 \leftrightarrow \ell_2]$ of arity 1 for all pairs of distinct labels, with the following semantics:

$$\begin{aligned} \{\mathtt{v}\}[\ell_1 \leftrightarrow \ell_2] &\xrightarrow{\delta} \mathtt{v} \\ \{\mathtt{w} \text{ with } \ell = \mathtt{v}\}[\ell_1 \leftrightarrow \ell_2] &\xrightarrow{\delta} \{\mathtt{w}[\ell_1 \leftrightarrow \ell_2] \text{ with } \ell = \mathtt{v}\} && \text{if } \ell \notin \{\ell_1, \ell_2\} \\ \{\mathtt{w} \text{ with } \ell = \mathtt{v}\}[\ell_1 \leftrightarrow \ell_2] &\xrightarrow{\delta} \{\mathtt{w}[\ell_1 \leftrightarrow \ell_2] \text{ with } \bar{\ell} = \mathtt{v}\} && \text{if } \{\ell, \bar{\ell}\} = \{\ell_1, \ell_2\} \end{aligned}$$

The initial environment Γ_0 must be extended with the following typing asumption:

$$\cdot[\ell_1 \leftrightarrow \ell_2] : \quad \forall \mathtt{X}_1 \mathtt{X}_2 \mathtt{Y}.\, \Pi(\ell_1 : \mathtt{X}_1 \ ; \ell_2 : \mathtt{X}_2 \ ; \mathtt{Y}) \to \Pi(\ell_1 : \mathtt{X}_2 \ ; \ell_2 : \mathtt{X}_1 \ ; \mathtt{Y})$$

We must then check subjection reduction for the new primitive. Since we only added a constructor, it suffices to check progress for the new primitive, that is, that well-typed expressions of the form $[\ell_1 \leftrightarrow \ell_2]\mathtt{v}_1 \ \ldots \ \mathtt{v}_n$ are either value or can be further reduced. Both parts are easy and similar to the corresponding parts in Exercice 1.11.16.

1.11.18 SOLUTION: There are several solutions. One of them is to asume a fixed total ordering on row-labels, and to retain as constructors only $\ell^{\kappa,L}$ such that $\ell < L$, that is $\ell < \ell'$ for all $\ell' \in L$; other constants $\ell^{\kappa,L}$ such that $\ell \not< L$ are moved from constructors to the status of destructors with the following collection of reduction rules:

$$\{\{\mathtt{w} \text{ with } \ell' = \mathtt{v}'\} \text{ with } \ell = \mathtt{v}\} \xrightarrow{\delta} \{\{\mathtt{w} \text{ with } \ell = \mathtt{v}\} \text{ with } \ell' = \mathtt{v}'\} \qquad (\text{RD-TRANSPOSE})$$

for all labels ℓ and ℓ' such that $\ell' < \ell$ and

$$\{\{\mathtt{w} \text{ with } \ell = \mathtt{v}'\} \text{ with } \ell = \mathtt{v}\} \xrightarrow{\delta} \{\mathtt{w} \text{ with } \ell = \mathtt{v}\} \qquad (\text{RD-DISCARD})$$

for all labels ℓ. It is now obvious that values are in normal forms, in the sense that explicit fields are never repeated and are always listed in order. Typing rules need not be changed, so requirement (i) of Definition 1.7.6 still holds. Requirement (ii) need to be check, in particular, for the new primitives ℓ^L, which we leave to the reader (the proof for $\cdot.\{\ell\}$ should hold unchanged).

1.11.19 SOLUTION: Let map have type $\Pi(\mathtt{X} \to \mathtt{Y}) \to \Pi(\mathtt{X}) \to \Pi(\mathtt{Y})$, and the following reduction rules in the semantics with normal forms:

$$\begin{aligned} \text{map } \{\mathtt{v}' \text{ with } \ell = \mathtt{v}\}\ \mathtt{w} &\xrightarrow{\delta} \{\text{map } \mathtt{v}'\ \mathtt{w} \text{ with } \ell = \mathtt{v}\ (\mathtt{w}.\{\ell\})\} \\ \text{map } \mathtt{v}\ \{\mathtt{w}' \text{ with } \ell = \mathtt{w}\} &\xrightarrow{\delta} \{\text{map } \mathtt{v}\ \mathtt{w}' \text{ with } \ell = (\mathtt{v}.\{\ell\})\ \mathtt{w}\} \\ \text{map } \{\mathtt{v}\}\ \{\mathtt{w}\} &\xrightarrow{\delta} \{\mathtt{v}\ \mathtt{w}\} \end{aligned}$$

1.11.22 SOLUTION: To ensure that the field is not present in the argument of extension, it sufficies to restrict its the typing asumptions as follows:

$$\langle \cdot \;\mathsf{with}\; \ell = \cdot \rangle : \forall \mathtt{X}\mathtt{X}'\mathtt{Y}.\; \Pi(\ell : \mathsf{abs} \;;\; \mathtt{Y}) \to \mathtt{X}' \to \Pi(\ell : \mathsf{pre}\;\; \mathtt{X}' \;;\; \mathtt{Y}).$$

To remove an existing field, we can use the following syntactic sugar:

$$\begin{aligned} \cdot \setminus \ell &\overset{\text{def}}{=} \lambda \mathtt{v}.\{\mathtt{v} \;\mathsf{with}\; \ell = \mathsf{abs}\} \\ &\quad : \forall \mathtt{X}\mathtt{Y}.\; \Pi(\ell : \mathtt{X} \;;\; \mathtt{Y}) \to \Pi(\ell : \mathsf{abs} \;;\; \mathtt{Y}) \end{aligned}$$

The following weaker typing asumption could also be used to ensure that the field is always present before removal:

$$\forall \mathtt{X}\mathtt{Y}.\; \Pi(\ell : \mathsf{pre}\;\; \mathtt{X} \;;\; \mathtt{Y}) \to \Pi(\ell : \mathsf{abs} \;;\; \mathtt{Y})$$

1.11.25 SOLUTION: The proof is similar to 1.11.16 but slightly more complex because we must also check that labels are defined when accessed, and with subtyping.

We reason simultaneously in both the subtyping model or the equal-only model, that is, we only rely on properties that are valid in both models.

We must first ensure that rules RE-FOUND and RE-FOLLOW respect (Definition 1.7.5).

∘ *Case* RE-FOUND: See Exercice **??**. In line **??**, field ℓ is $\mathsf{pre}\;\mathtt{X}_1$ instead of $\mathtt{X}_1$ and $\mathsf{pre}\;\;\mathtt{Y}_2$ instead of $\mathtt{Y}_2$ and step **??** also uses covariance of pre.

∘ *Case* RE-FOLLOW The proof is similar.

We must then check that if the configuration $F\;\mathtt{v}_1\;\ldots\;\mathtt{v}_k/\mu$ is is well-typed, then either it is reducible, or it is a value.

We begin by checking that every value that is well-typed with type $\Pi\,\mathtt{T}$ is a record value, that is, either of the form $\langle\rangle$ or $\langle \mathtt{v}'' \;\mathsf{with}\; \ell' = \mathtt{v}' \rangle$. See Exercice 1.11.16.

Next, we note that, according to the constraint generation rules, if the configuration $\mathsf{c}\,\mathtt{v}_1\;\ldots\;\mathtt{v}_k/\mu$ is well-typed, then a constraint of the form $\mathsf{let}\;\Gamma_0; \mathsf{ref}\,M\;\mathsf{in}\;(\mathsf{c} \preceq \mathtt{X}_1 \to \ldots \to \mathtt{X}_k \to \mathtt{T} \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!] \wedge \ldots \wedge [\![\mathtt{v}_k : \mathtt{X}_k]\!])$ is satisfiable. We now reason by cases on c.

∘ *Case* c is $\langle\rangle$ or $\langle \cdot \;\mathsf{with}\; \ell = \cdot \rangle$. See Exercice 1.11.16.

∘ *Case* c is $\cdot.\langle\ell\rangle$. We may asume $k \geq 1$, since otherwise, the expression is a value. Then $\Gamma_0(\mathsf{c})$ is $\forall \mathtt{X}\mathtt{Y}.\Pi(\ell : \mathsf{pre}\;\;\mathtt{X} \;;\; \mathtt{Y}) \to \mathtt{X}$, so by C-INID and C-ARROW the above constraint entails $\mathsf{let}\;\Gamma_0; \mathsf{ref}\,M\;\mathsf{in}\;(\exists \mathtt{X}\mathtt{Y}.(\mathtt{X}_1 \leq \Pi(\ell : \mathsf{pre}\;\;\mathtt{X} \;;\; \mathtt{Y})) \wedge [\![\mathtt{v}_1 : \mathtt{X}_1]\!])$, which by lemma 1.6.3 entails $\mathsf{let}\;\Gamma_0; \mathsf{ref}\,M\;\mathsf{in}\;\exists \mathtt{X}\mathtt{Y}.[\![\mathtt{v}_1 : \Pi(\ell : \mathsf{pre}\;\;\mathtt{X} \;;\; \mathtt{Y})]\!]$. Thus $\mathtt{v}_1$ is a record value, that is, either of the form $\langle\rangle$ or $\langle \mathtt{v}'' \;\mathsf{with}\; \ell = \mathtt{v}' \rangle$. In fact, the former case cannot occur, since $\mathsf{let}\;\Gamma_0; \mathsf{ref}\,M\;\mathsf{in}\;\exists \mathtt{X}\mathtt{Y}.[\![\langle\rangle : \Pi(\ell : \mathsf{pre}\;\;\mathtt{X} \;;\; \mathtt{Y})]\!]$ entails $\exists \mathtt{X}\mathtt{Y}\Pi(\partial\mathsf{abs}) \leq \Pi(\ell : \mathsf{pre}\;\;\mathtt{X} \;;\; \mathtt{Y})$ by C-INID and C-IN*, which in turns entails $\exists \mathtt{X}.\mathsf{abs} \leq \mathsf{pre}\ \mathtt{X}$ by C-ROW-DL and covariance of Π and ℓ. However, this constraint is equivalent to false, because $\phi(\mathsf{abs}) \leq \phi(\mathsf{pre}\ \mathtt{X})$ does not hold in any ground assignment ϕ. Thus $\mathtt{v}_1$ is $\langle \mathtt{v}'' \text{ with } \ell' = \mathtt{v}' \rangle$ and the configuration is reducible to $\mathtt{v}'$ if ℓ' is ℓ or $\mathtt{v}''$ otherwise.