Chapter 2

A type assignment system[1]

In this chapter, we develop a system of type assignment with intersection types, union types, indexed types, and universal and existential dependent types that is sound in a call-by-value functional language. The combination of logical and computational principles underlying our formulation naturally leads to the central idea of typechecking subterms in evaluation order. We thereby provide a uniform generalization and explanation of several earlier isolated systems. While undecidable, this system is the basis for the decidable system in Chapter 3, the practical system in Chapter 5, and ultimately the typechecker described in Chapter 6.

The system in this chapter is a Curry-style type assignment system. Our goal is to typecheck code, not to compile it. Elaborating the source language into a form suitable for use as a typed intermediate language—one with explicit polymorphic instantiation, for example—would have few, if any, advantages in this typechecking-focused setting. In this respect, we follow Davies' datasort refinement system [Dav05a], which is also focused on typechecking rather than compilation. Thus, we see no compelling arguments against type assignment in this setting, and prior work suggests that it is a solid foundation for type refinement systems.

2.1 Introduction

Conventional static type systems are tied directly to the expression constructs available in a language. For example, functions are classified by function types $A \to B$, pairs are classified by product types $A * B$, and so forth. In more advanced type systems we find type constructs that are independent of any particular expression construct. The best-known examples are parametric polymorphism $\forall\alpha.\, A$ and intersection polymorphism $A \wedge B$. Such types can be seen as expressing more complex properties of programs. For example, if we read the judgment $e : A$ as *e satisfies property A*, then $e : A \wedge B$ expresses that e satisfies both property A and property B. We call such types *property types*. The aim is to integrate a rich system of property types into practical languages such as Standard ML [MTHM97], in order to express and verify detailed invariants of programs as part of typechecking.

In this chapter we design a system of property types specifically for call-by-value languages.

[1] This chapter is based on joint work with Frank Pfenning [DP03].

We show that the resulting system is type-safe, that is, satisfies the type preservation and progress theorems. We include indexed types $\delta(i)$, intersection types $A \wedge B$, a greatest type $\top$, universal dependent types $\Pi a{:}\gamma.\, A$, union types $A \vee B$, an empty type $\bot$, and existential dependent types $\Sigma a{:}\gamma.\, A$. We thereby combine, unify, and extend prior work on intersection types [DP00], union types [Pie91b, BDCd95] and index refinements [Xi98, XP99].

Several ideas emerge from our investigation. Perhaps most important is that type assignment may visit subterms in evaluation order, rather than just relying on immediate subterms. We also confirm the critical importance of a logically motivated design for subtyping and type assignment. The resulting orthogonality of various property type constructs greatly simplifies the theory and allows one to understand each concept in isolation. As a consequence, simple types, intersection types [DP00], and indexed types [XP99] are extended *conservatively*. The type system is designed to allow effects (in particular, mutable references), but in order to concentrate on more basic issues, we do not include them explicitly (see [DP00] for the applicable techniques to handle mutable references).

The system of pure type assignment presented in this chapter is undecidable. Chapter 3 presents a decidable version based on bidirectional typechecking (in the style of [DP00, Dun02]) of programs containing some type annotations, where we variously *check* an expression against a type or else *synthesize* the expression's type.

The remainder of the chapter is organized as follows. We start by defining a small and conventional functional language with subtyping, in a standard call-by-value semantics. We then add several forms of property types: intersection types, indexed types, and universal dependent types. As we do so, we motivate our typing and subtyping rules through examples, showing how our particular formulation arises out of our demand that the theorems of *type preservation* and *progress* hold. Then we add the *indefinite* property types: the empty type $\bot$, the union type $\vee$, and the existential dependent type Σ. To be sound, these must visit subterms in evaluation order. After proving some novel properties of judgments and substitutions, we prove preservation and progress. Finally, we discuss related work and conclude.

2.2 The base language

We start by defining a standard call-by-value functional language (Figure 2.1) with functions, a unit type (used in a few examples), products, and recursion, to which we will add various constructs and types. Expressions do not contain types, because we are formulating a pure type assignment system. We distinguish between variables x that stand for values and variables u that stand for expressions, where the latter arise only from fixed points **fix** $u.\, e$. The form of the typing judgment is

$$\Gamma \vdash e : A$$

where Γ is a context typing variables x and u. The typing rules so far are standard (Figure 2.2); the subsumption rule utilizes a subtyping judgment

$$\Gamma \vdash A \leq B$$

meaning that A is a subtype of B in context Γ. The interpretation is that the set of values of type A is a subset of the set of values of type B. The context Γ is not used in the subtyping rules of Figure

$$A, B, C, D ::= \mathbf{1} \mid A \to B \mid A * B$$
$$e ::= x \mid u \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{fix}\, u.\, e \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e)$$

Figure 2.1: Syntax of types and terms in the initial language

$$\frac{\Gamma \vdash B_1 \le A_1 \qquad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 \to A_2 \le B_1 \to B_2}\to \qquad \frac{}{\Gamma \vdash \mathbf{1} \le \mathbf{1}}\,\mathbf{1} \qquad \frac{\Gamma \vdash A_1 \le B_1 \qquad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 * A_2 \le B_1 * B_2}*$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\,\mathsf{var} \qquad \frac{\Gamma(u) = A}{\Gamma \vdash u : A}\,\mathsf{fixvar} \qquad \frac{\Gamma, u{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\, u.\, e : A}\,\mathsf{fix}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e : B}\,\mathsf{sub}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\to\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\to\mathrm{E}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\,\mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 * A_2}*\mathrm{I} \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{fst}(e) : A}*\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{snd}(e) : B}*\mathrm{E}_2$$

Figure 2.2: Subtyping and typing in the initial language

2.2, but we subsequently augment the subtyping system with rules that refer to Γ. The rule $\to$ is the standard subtyping rule for function types, contravariant in the argument and covariant in the result; rule **1** is obvious. It is easy to prove that subtyping is decidable, reflexive ($\Gamma \vdash A \le A$), and transitive (if $\Gamma \vdash A \le B$ and $\Gamma \vdash B \le C$ then $\Gamma \vdash A \le C$); as we add rules to the subtyping system, we maintain these properties.

The subtyping rules for our system are designed following the well-known principle that $A \le B$ only if any (closed) value of type A also has type B. Thus, whenever we must check if an expression e has type B we are safe if we can synthesize a type A and $A \le B$. The subtyping rules then naturally decompose the structure of A and B by so-called *left* and *right* rules that closely mirror the rules of a sequent calculus [Pra65, Appendix A].

A call-by-value operational semantics defining a relation $e \mapsto e'$ is given in Figure 2.3. We use v for values, and write e value to mean that e is a value. We write $\mathcal{E}$ for an evaluation context—a term containing a hole $[]$; we write $\mathcal{E}[e']$ to denote $\mathcal{E}$ with its hole replaced by e'.

2.3 Definite property types

Definite types accumulate positive information about expressions. For instance, the intersection type $A \wedge B$ expresses the conjunction of the properties A and B. We later introduce *indefinite types* such as $A \vee B$ which encompass expressions that have either property A or property B, although it is unknown which one.

$$\text{Values} \quad v ::= x \mid () \mid \lambda x.\, e \mid (v_1, v_2)$$
$$\text{Evaluation contexts} \quad \mathcal{E} ::= [] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) \mid \mathbf{fst}(\mathcal{E}) \mid \mathbf{snd}(\mathcal{E})$$

$$\frac{e' \mapsto_R e''}{\mathcal{E}[e'] \mapsto \mathcal{E}[e'']}\,\mathsf{ev\text{-}context}$$

$$\begin{aligned}(\lambda x.\, e)\, v &\mapsto_R [v/x]\, e\\ \mathbf{fix}\, u.\, e &\mapsto_R [(\mathbf{fix}\, u.\, e)\,/\,u]\, e\\ \mathbf{fst}(v_1, v_2) &\mapsto_R v_1\\ \mathbf{snd}(v_1, v_2) &\mapsto_R v_2\end{aligned}$$

Figure 2.3: A small-step call-by-value semantics

$$\begin{aligned}ms &::= \cdot \mid c(x) \Rightarrow e \mid ms\\ e &::= \ldots \mid c(e) \mid \mathbf{case}\ e\ \mathbf{of}\ ms\\ v &::= \ldots \mid c(v)\\ \mathcal{E} &::= \ldots \mid c(\mathcal{E}) \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ ms\end{aligned}$$
$$\mathbf{case}\ c(v)\ \mathbf{of}\ \ldots c(x) \Rightarrow e \ldots \mapsto_R [v/x]\, e$$

Figure 2.4: Extending the language with datatypes

2.3.1 Refined datatypes

We now add datatypes with refinements (Figure 2.4). $c(e)$ denotes a datatype constructor c applied to an argument e; the destructor **case** e **of** ms denotes analysis of e with one layer of non-redundant and exhaustive matches ms, each of the form $c_k(x_k) \Rightarrow e_k$. For example, the only permitted **case** expression on the list type is **case** e **of** $\mathsf{Nil}(x_1) \Rightarrow e_1 \mid \mathsf{Cons}(x_2) \Rightarrow e_2$, in which $x_1 : \mathbf{1}$ and $x_2 : \mathsf{int}*\mathsf{list}$. This restricted language of patterns will be enlarged in Chapter 4.

Each datatype is refined by an *atomic subtyping* relation $\preceq$ over *datasorts* δ. Each datasort identifies a subset of values of the form $c(v)$, yielding definite information about a value. For example, datasorts true and false identify singleton subsets of values of the type bool.

A new subtyping rule defines subtyping for datasorts in terms of the atomic subtyping relation $\preceq$:

$$\frac{\delta_1 \preceq \delta_2}{\Gamma \vdash \delta_1 \le \delta_2}\,\delta$$

To maintain reflexivity and transitivity of subtyping, we require the same properties of atomic subtyping: $\preceq$ must be reflexive and transitive.

Since we will subsequently further refine our datatypes by indices, we defer discussion of the typing rules.

2.3.2 Intersections

The typing $e : A \wedge B$ expresses that e has type A and type B. The subtyping rules for $\wedge$ capture this:

$$\frac{\Gamma \vdash A \leq B_1 \quad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\wedge R \qquad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\wedge L_1 \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\wedge L_2$$

We omit the common distributivity rule

$$\overline{(A \to B) \wedge (A \to B') \leq A \to (B \wedge B')}$$

which Davies and Pfenning showed to be unsound in the presence of mutable references [DP00]. The present system does not have mutable references, but is intended to be compatible with such, as well as with other forms of effectful computation; Davies [Dav05a, pp. 54–55] analyzes distributivity in the more general context of effects captured by the $\bigcirc$ modality [Mog88]. Another reason to omit the distributivity rule is that without it, no subtyping rule contains more than one type constructor: the rules are orthogonal. As we add type constructors and subtyping rules, we maintain this orthogonality. In fact, the left and right subtyping rules for $\wedge$ and the other property types closely mirror the left and right rules of a sequent calculus [Pra65, Appendix A]. Ignoring Γ, we can think of subtyping as a single-antecedent, single-succedent form of the sequent calculus.

On the level of typing, we can introduce an intersection with the rule

$$\frac{\Gamma \vdash v : A_1 \qquad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}\wedge I$$

and eliminate it with

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\wedge E_1 \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\wedge E_2$$

Note that $\wedge$I can only type values v, not arbitrary expressions, following Davies and Pfenning [DP00] who showed that in the presence of mutable references, allowing non-values destroys type preservation.

The $\wedge$-elimination rules are derivable via rule sub with the $\wedge L_{1,2}$ subtyping rules. However, we include them because they are not derivable in a bidirectional system such as that of Chapter 3.

2.3.3 Greatest type: $\top$

It is easy to incorporate a greatest type $\top$, which can be thought of as the 0-ary form of $\wedge$. The rules are simply

$$\frac{}{\Gamma \vdash A \leq \top}\top R \qquad \frac{\Gamma \vdash v \text{ ok}}{\Gamma \vdash v : \top}\top I$$

There is no left subtyping rule. The typing rule is essentially the 0-ary version of $\wedge$I, the rule for binary intersection. The premise $\Gamma \vdash v$ ok says that the free variables of e are in $\mathrm{dom}(\Gamma)$. Without this premise, we could derive $\cdot \vdash y : \top$, where y is an unknown identifier. Such anomalies would be problematic in Chapter 5. Finally, note that if we allowed $\top$I to type non-values, the progress theorem would fail: $\vdash ()() : \top$, but $()()$ is neither a value nor a redex.

$$\begin{array}{lll} P & ::= & \bot \mid i \doteq j \mid \ldots \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, u{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P \end{array} \qquad \begin{array}{rcl} \overline{\cdot} & = & \cdot \\ \overline{\Gamma, x{:}A} & = & \overline{\Gamma} \\ \overline{\Gamma, a{:}\gamma} & = & \overline{\Gamma}, a{:}\gamma \\ \overline{\Gamma, P} & = & \overline{\Gamma}, P \end{array}$$

Figure 2.5: Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

$$\boxed{\Gamma \vdash A \text{ wf}} \qquad \frac{FV(A) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash A \text{ wf}} \qquad \boxed{\Gamma \vdash P \text{ wf}} \qquad \frac{FV(P) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash P \text{ wf}}$$

$$\boxed{\Gamma \text{ wf}} \qquad \frac{}{\cdot \text{ wf}} \qquad \frac{\Gamma \text{ wf} \qquad x \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash A \text{ wf}}{\Gamma, x{:}A \text{ wf}}$$

$$\frac{\Gamma \text{ wf} \qquad a \notin \mathrm{dom}(\Gamma)}{\Gamma, a{:}\gamma \text{ wf}} \qquad \frac{\Gamma \text{ wf} \qquad \Gamma \vdash P \text{ wf}}{\Gamma, P \text{ wf}}$$

Figure 2.6: Well-formedness of types, propositions, and contexts

2.3.4 Index refinements and universal dependent types Π

Now we add index refinements, which are dependent types over a restricted domain, closely following Xi and Pfenning [XP99], Xi [Xi98, Xi00], and Dunfield [Dun02]. This refines datatypes not only by datasorts, but by indices drawn from some constraint domain: the type $\delta(i)$ is the refinement by δ and index i.

To accommodate index refinements, several changes must be made to the systems we have constructed so far. The most drastic is that Γ can include *index variables* a, b and propositions P as well as program variables. Because the program variables are irrelevant to the index domain, we can define a *restriction function* $\overline{\Gamma}$ that yields its argument Γ without program variable typings (Figure 2.5). No variable may appear twice in Γ, but ordering of the variables is now significant because of dependencies. That is, in the context $\Gamma_1, x{:}\mathsf{list}(a), \Gamma_2$, the index variable a must be declared ($a{:}\gamma$) in Γ_1—contexts such as $x{:}\mathsf{list}(a), a{:}\mathcal{N}$ are ill-formed. These requirements are made explicit in rules for the *well-formedness* of types, propositions, and contexts (Figure 2.6). However, we do not refer to these rules except implicitly, as we assume throughout the thesis that all types, propositions, and contexts are well formed. For instance, given the context $\Gamma_1, a{:}\gamma, \Gamma_2$, we know from the implicit $(\Gamma_1, a{:}\gamma, \Gamma_2)$ wf that $a \notin \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$ and $a \notin FV(\Gamma_1)$.

Our formulation, like Xi's, requires only a few properties of the constraint domain: There must be a way to decide a consequence relation

$$\overline{\Gamma} \models P$$

whose interpretation is that given the index variable typings and propositions in $\overline{\Gamma}$, the proposition

Property 2.2 (Substitution). *Given* $\overline{\Gamma_1} \vdash i : \gamma$*:*

(i) If $\overline{\Gamma_1}, a{:}\gamma, \overline{\Gamma_2} \models P$ *then* $\overline{\Gamma_1}, [i/a]\overline{\Gamma_2} \models [i/a]P$.

(ii) If $\overline{\Gamma_1}, a{:}\gamma, \overline{\Gamma_2} \vdash j : \gamma'$ *then* $\overline{\Gamma_1}, [i/a]\overline{\Gamma_2} \vdash [i/a]j : \gamma'$.

Property 2.3 (Weakening). *Given* $(\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2})$ wf*:*

(i) If $\overline{\Gamma_1}, \overline{\Gamma_2} \vdash i : \gamma$ *then* $\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2} \vdash i : \gamma$.

(ii) If $\overline{\Gamma_1}, \overline{\Gamma_2} \models P$ *then* $\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2} \models P$.

Property 2.4 (Equivalence). *If* $\overline{\Gamma} \vdash i : \gamma$ *and* $\overline{\Gamma} \vdash j : \gamma$ *and* $\overline{\Gamma} \vdash k : \gamma$ *then*

(i) The relation $\overline{\Gamma} \models i \doteq i$ *holds.*

(ii) If $\overline{\Gamma} \models i \doteq j$ *then* $\overline{\Gamma} \models j \doteq i$.

(iii) If $\overline{\Gamma} \models i \doteq j$ *and* $\overline{\Gamma} \models j \doteq k$*, then* $\overline{\Gamma} \models i \doteq k$.

Property 2.5. *The relation* $\cdot \models \bot$ *does not hold.*

Property 2.6. $\overline{\Gamma}, a{:}\gamma \vdash a : \gamma$.

Property 2.7 (Consequence). *If* $\overline{\Gamma} \models P_k$ *for all* $P_k \in \{P_1, \ldots, P_n\}$*, and* $\overline{\Gamma}, P_1, \ldots, P_n \models P'$ *then* $\overline{\Gamma} \models P'$.

Figure 2.7: Assumed properties of the $\models$ and $\vdash$ index relations.

P must hold. Among the propositions must be $i \doteq j$, denoting equality. There must be a way to decide a relation

$$\overline{\Gamma} \vdash i : \gamma$$

whose interpretation is that i has *sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have sorts; terms and indices are distinct. Our proofs require that $\models$ be a consequence relation (that is, if some assumption in $\overline{\Gamma}$ is entailed by the rest of $\overline{\Gamma}$, it can be removed without changing what is entailed), that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that both $\models$ and $\vdash$ have obvious substitution and weakening properties (Figure 2.7).

Remark 2.1. The system in this chapter is undecidable, so there is no fundamental reason any of these relations must be decidable; the requirement becomes necessary in Chapter 3.

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. In our examples, we work in a domain of integers $\mathcal{N}$ with $\doteq$ and some standard operations ($+$, $-$, $*$, $<$, and so on); each datatype is refined by indices of sort $\mathcal{N}$. Then $\overline{\Gamma} \models P$ is decidable provided the inequalities in P are linear.

We add an infinitary definite type $\Pi a{:}\gamma.\, A$, introducing an index variable a universally quantified over indices of sort γ. One can also view Π as a dependent product restricted to indices (instead of arbitrary terms). We also add a *guarded type* $P \supset A$, read "P implies A", discussed below.

Example. Assume we define a datatype of integer lists: a list is either $\mathsf{Nil}()$ or $\mathsf{Cons}(h, t)$ for some integer h and list t. Refine this type by a datasort odd of odd-length lists, and by a datasort even of even-length lists. We also refine the lists by their length, so Nil has type $\mathbf{1} \to \mathsf{even}(0)$, and Cons has type $(\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{even}(a) \to \mathsf{odd}(a+1)) \wedge (\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{odd}(a) \to \mathsf{even}(a+1))$. Then the function

$$\textbf{fix}\ \mathit{repeat}.\ \lambda x.\ \textbf{case}\ x\ \textbf{of}\ \mathsf{Nil}() \Rightarrow \mathsf{Nil}()\ \mid\ \mathsf{Cons}(h,t) \Rightarrow \mathsf{Cons}(h, \mathsf{Cons}(h, \mathit{repeat}(t)))$$

will have type $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$.

To handle the indices, we modify the subtyping rule δ from Section 2.3.1 so that it checks (separately) the datasorts δ_1, δ_2 and the indices i, j:

$$\frac{\delta_1 \preceq \delta_2 \qquad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ \delta$$

Datatype constructors c are typed by a judgment

$$\Gamma \vdash c : A^{con}$$

where

$$\text{Constructor types}\quad A^{con} ::= B \to \delta(i) \mid A^{con} \wedge A^{con} \mid \Pi a{:}\gamma.\, A^{con} \mid P \supset A^{con}$$

That is, the type of a constructor is $B \to \delta(i)$ where B is unrestricted, or an intersection, universal quantification, or guard of such a type. The only rule deriving $\Gamma \vdash c : A^{con}$ is $\mathcal{S}$-con, which uses a *synthesis subtyping* judgment $\Gamma \vdash A \uparrow B$ (see Figure 2.8). Synthesis subtyping is essentially a weaker form of subtyping with A as input and B as output: if $A \uparrow B$ then $A \leq B$, but not the converse. For example, $A \leq \top$ but $A \not\uparrow \top$.

We assume a *constructor signature* $\mathcal{S}$ that gives types to constructors. In the implementation, this is given explicitly by the user; see Chapter 6.[2]

The rule for constructor application is

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \qquad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\ \delta\mathrm{I}$$

To type **case** e **of** ms where $e : \delta(i)$, rule δE checks that all the case arms in ms have the same type C, expressed by the premise $\Gamma \vdash ms :_{\delta(i)} C$, read "$ms$ checks against C, assuming the expression cased upon has type $\delta(i)$".

$$\frac{\Gamma \vdash e : \delta(i) \qquad \Gamma \vdash ms :_{\delta(i)} C}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms : C}\ \delta\mathrm{E}$$

To check an arm $c(x) \Rightarrow e$, we analyze the type $\mathcal{S}(c)$ of the constructor, accounting for all the ways c could have been applied to create a value of type $\delta(i)$. For example, if $\mathcal{S}(c) = (A_1 \to \delta_1(i_1)) \wedge (A_2 \to \delta(i_2))$, we need to check $e : C$ with two obligations: in the first we assume $x{:}A_1, i_1 \doteq i$, and in the second $x{:}A_2, i_2 \doteq i$. We introduce a judgment form

$$\Gamma; c : A^{con}; c(x) : B \vdash e : C$$

[2] This approach differs materially from Davies' system [Dav05a], which *generates* constructor types from a regular tree grammar; see Section 7.4.7.

$\boxed{\Gamma \vdash A \uparrow B}$

$$\frac{}{\Gamma \vdash A \uparrow A}\ \text{refl-}\uparrow \qquad \frac{\Gamma \vdash A \uparrow A'}{\Gamma \vdash A \wedge B \uparrow A'}\ \wedge\uparrow_1 \qquad \frac{\Gamma \vdash B \uparrow B'}{\Gamma \vdash A \wedge B \uparrow B'}\ \wedge\uparrow_2$$

$$\frac{\Gamma \vdash [i/a]A \uparrow A' \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \uparrow A'}\ \Pi\uparrow \qquad \frac{\Gamma \vdash A \uparrow A' \qquad \overline{\Gamma} \models P}{\Gamma \vdash P \supset A \uparrow A'}\ \supset\uparrow$$

$\boxed{\Gamma \vdash c : A^{con}}$

$$\frac{\Gamma \vdash \mathcal{S}(c) \uparrow A}{\Gamma \vdash c : A}\ \mathcal{S}\text{-con}$$

Figure 2.8: Constructor typing

which is read "under Γ, assuming c has the constructor type A^{con} and $c(x)$ (which is the value cased upon) has type B, the case arm e has type C".

$$\frac{}{\Gamma \vdash \cdot :_B C}\ \mathsf{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e : C \qquad \Gamma \vdash ms :_B C}{\Gamma \vdash (c(x) \Rightarrow e \mid ms) :_B C}\ \mathsf{casearm}$$

Rules δS-ct and δF-ct cover the "base case" in which the type of c is simply $A \to \delta(i)$. In each rule, we have assumptions that a constructor c has type $A \to \delta(i)$ and a value $c(x)$ has type $\delta'(i')$.

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i' \vdash e : C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta' \qquad \Gamma, x{:}A \vdash e\ \mathsf{ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{F-ct}$$

If $\delta \not\preceq \delta'$ (rule δF-ct), those assumptions are inconsistent: either

- δ and δ' are incomparable (neither $\delta \preceq \delta'$ nor $\delta' \preceq \delta$), so clearly $c : A \to \delta(i)$ cannot produce a $\delta'(i')$ when applied to x, or
- $\delta' \preceq \delta$ (but $\delta \not\preceq \delta'$); here, observe that $c : A \to \delta(i)$ represents *all* the information we have about the result of c—it cannot be the case that $c(x)$ "really" has the smaller datasort δ'—so the assumption $c(x) : \delta'(i)'$ is just as inconsistent as $1 \doteq 2$ is,

so in that case we need not examine e at all.

Remark 2.8. The second point may not be obvious. Suppose we have datasorts nonempty and list such that nonempty $\preceq$ list, but of course list $\not\preceq$ nonempty, and $\mathcal{S}(\mathsf{Nil}) = \mathbf{1} \to \mathsf{list}$. Given an $e' : \mathsf{nonempty}$, to derive **case** e' **of** $\mathsf{Nil} \Rightarrow e : C$ we try to derive

$$\Gamma; \mathsf{Nil} : \mathbf{1} \to \mathsf{list}; \mathsf{Nil}(x) : \mathsf{nonempty} \vdash e : C.$$

Since list $\not\preceq$ nonempty, we can apply rule δF-ct. The judgment expresses the assumptions that Nil has type $\mathbf{1} \to$ list and a particular value $\mathsf{Nil}(x)$ has type nonempty. If these assumptions were consistent, rule δF-ct would be wrong! But it cannot be the case that $\mathsf{Nil}(x) : \mathsf{nonempty}$. We assume $\mathsf{Nil} : \mathbf{1} \to \mathsf{list}$, given to us by the user's constructor signature $\mathcal{S}$; we have no other information about

Nil. We cannot possibly show that a value of type list "really" also has type nonempty. Therefore the assumptions are inconsistent.

However, if $\delta \preceq \delta'$ (rule δS-ct), either $\delta' \preceq \delta$ (morally, $\delta' = \delta$) which clearly makes the subsorting assumptions consistent, or $\delta' \not\preceq \delta$ (morally, $\delta \prec \delta'$) in which case $c(x) : \delta'(i')$ is not very specific—some information about the value was lost between its construction and its deconstruction-by-**case**—but is still consistent. Either way, we must check $e : C$ assuming that $x{:}A$ and $i \doteq i'$ hold. The latter can make the context inconsistent: if $c : A \to \delta(0)$ and $c(x) : \delta(3)$, we can obtain $\Gamma, x{:}A, 0 \doteq 3 \vdash e : C$ through a rule contra:

$$\frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e\ \mathsf{ok}}{\Gamma \vdash e : A}\ \mathsf{contra}$$

The bodies of impossible case arms are thus skipped: by δF-ct if the datasort is impossible, by contra if the index is impossible. The remaining rules are for the types $\wedge$, Π, and $\supset$, and recapitulate—as left rules—the structure of the corresponding introduction rules. For example, $\wedge$-ct moves from the assumption $c : A_1^{con} \wedge A_2^{con}$, in the conclusion, to $c : A_1^{con}$ in the first premise and $c : A_2^{con}$ in the second, just as $\wedge$I moves from $\cdots : A_1 \wedge A_2$ in its conclusion to $\cdots : A_1$ and $\cdots : A_2$ in its premises.

$$\frac{\Gamma; c : A_1^{con}; c(x) : B \vdash e : C \qquad \Gamma; c : A_2^{con}; c(x) : B \vdash e : C}{\Gamma; c : A_1^{con} \wedge A_2^{con}; c(x) : B \vdash e : C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : \Pi a{:}\gamma.\, A^{con}; c(x) : B \vdash e : C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : (P \supset A^{con}); c(x) : B \vdash e : C}\ \supset\text{-ct}$$

The subtyping rules for Π are

$$\frac{\Gamma \vdash [i/a]A \leq B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ \Pi\mathrm{L} \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ \Pi\mathrm{R}$$

The left rule allows one to instantiate a quantified index variable a to an index i of appropriate sort. The right rule states that if $A \leq B$ regardless of an index variable b, A is also a subtype of $\Pi b{:}\gamma.\, B$. Of course, b cannot occur free in A.

The typing rules for Π are

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\ \Pi\mathrm{I} \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\ \Pi\mathrm{E}$$

Like $\wedge$I, and for similar reasons (to maintain type preservation), ΠI is restricted to values. Moreover, if γ is an empty sort, progress would fail if the rule were not thus restricted.

2.3.5 Guarded types

The *guarded type* $P \supset A$, read "P implies A", is equivalent to A if the proposition P holds, and is useless otherwise (as $\top$ is useless). To illustrate, suppose our index domain is the integers (index sort $\mathcal{Z}$) with propositions $P ::= i < j \mid i \leq j \mid \ldots$. Then

$$\Pi a{:}\mathcal{Z}.\, (a \geq 0) \supset (\mathsf{int}(a) \to \mathsf{list}(a))$$

is the type of functions from natural numbers to lists of the corresponding length: a function of this type can only be applied to arguments $\mathsf{int}(i)$ such that $i \geq 0$.[3]

The introduction and elimination rules are quite natural. Note that to ensure progress the introduction rule—like the introduction rules for other definite types—is restricted to values: otherwise, we could use contra and $\supset$I to show $\vdash ()() : (\bot \supset A)$, which is not a value but does not take a step.

$$\frac{\Gamma, P \vdash v : A}{\Gamma \vdash v : P \supset A} \supset I \qquad \frac{\Gamma \vdash e : P \supset A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : A} \supset E$$

The subtyping rules for $\supset$ are straightforward: reading *both* $\supset$ and $\leq$ as "implies", the left rule $\supset$L says that if P holds, and A implies B, then (P implies A) implies B. Reading $\vdash$ as "implies" as well, the right rule $\supset$R says that if P implies (A implies B), then A implies (P implies B).

$$\frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash (P \supset A) \leq B} \supset L \qquad \frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash A \leq (P \supset B)} \supset R$$

Xi's early work [Xi00, Xi98] achieved the effect of $P \supset A$ through a different mechanism, the *subset sort* $\{a{:}\gamma \mid P\}$, which allows a constraint to be placed on the sort:

$$\Pi a{:}\overbrace{\{a{:}\mathcal{Z} \mid a \geq 0\}}. \mathsf{int}(a) \to \mathsf{list}(a)$$

Recent work by Xi [Xi04] uses a form of guarded types (in a somewhat different setting). Our type system does not include subset sorts; however, our implementation permits them through a preprocessing phase described in Section 6.3.4. For example, the *repeat* function's type, $\Pi a{:}\mathcal{N}. \mathsf{list}(a) \to \mathsf{even}(2 * a)$, becomes $\Pi a{:}\mathcal{Z}. (a \geq 0) \supset (\mathsf{list}(a) \to \mathsf{even}(2 * a))$.

2.4 Indefinite property types

We now have a system with definite types $\wedge$, $\top$, Π, and $\supset$. The typing and subtyping rules are both orthogonal and internally regular: no rule mentions both $\top$ and $\wedge$, $\top$I is a 0-ary version of $\wedge$I, and so on. However, one cannot express the types of functions with indeterminate result type. A simple example is a *filter* f l function on lists of integers, which returns the elements of l for which u returns true. It has the ordinary type

$$\mathit{filter} : (\mathsf{int} \to \mathsf{bool}) \to \mathsf{list} \to \mathsf{list}$$

Indexing lists by their length, the refined type should look like[4]

$$\mathit{filter} : (\mathsf{int} \to \mathsf{bool}) \to \Pi n{:}\mathcal{N}. \mathsf{list}(n) \to \mathsf{list}(\_\_)$$

[3] In this and other examples, we use a singleton type $\mathsf{int}(i)$: every literal integer n has type $\mathsf{int}(n)$.

[4] In the past, we (and others) have given the refined type of *filter* with the Π on the outside, i.e. $\Pi n{:}\mathcal{N}. (\mathsf{int} \to \mathsf{bool}) \to \ldots$, which is quite unfortunate if *filter* is partially applied: n must be instantiated before *filter* is applied to its first argument, so in **let** *filter_f=filter* f **in** ..., the function *filter_f* can only be applied to lists known to all have the same length!

But we cannot fill in the blank. Xi's solution [XP99, Xi98] was to add dependent sums $\Sigma a{:}\gamma. A$ quantifying existentially over index variables. Then we can express the fact that *filter* returns a list of some indefinite length m as follows[5]:

$$\mathit{filter} : (\mathsf{int} \to \mathsf{bool}) \to \Pi n{:}\mathcal{N}. \mathsf{list}(n) \to (\Sigma m{:}\mathcal{N}. \mathsf{list}(m))$$

For similar reasons, we also occasionally need 0-ary and binary indefinite types—the empty type and union types, respectively. We begin with the binary case.

2.4.1 Unions

On values, the binary indefinite type should simply be a union in the ordinary sense: if $\vdash v : A \vee B$ then either $\vdash v : A$ or $\vdash v : B$. This leads to the following subtyping rules, which are dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B} \vee L \qquad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2} \vee R_1 \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2} \vee R_2$$

The introduction rules directly express the simple logical interpretation:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B} \vee I_2$$

The elimination rule is harder to formulate. It is clear that if $e : A \vee B$ and e evaluates to a value v, then either $v : A$ or $v : B$. So we should be able to reason by cases, similar to the usual disjunction elimination rule in natural deduction [Pra65, p. 20]. However, there are several complications. The first is that $A \vee B$ is a property type. That is, we cannot have a **case** construct in the ordinary sense since the members of the union are not tagged.[6]

As a simple example, consider

$$\begin{array}{lcl} f & : & (B \to D) \wedge (C \to D) \\ g & : & A \to (B \vee C) \\ x & : & A \end{array}$$

Then $f(g(x))$ should be type correct and have type D. At first this might seem doubtful, because the type of f does not directly show how to treat an argument of type $B \vee C$. However, whatever g returns must be a closed value v, and must therefore either have type B or type C. In both cases $f(v)$ should be well-typed and return a result of type D.

Note that we can distinguish cases on the result of $g(x)$ because it is evaluated *before* f is called.[7] In general, we allow case distinction on the type of the next expression to be evaluated. This guarantees both progress and preservation. The rule is then

[5] The additional constraint $m \leq n$ can be expressed by an *asserting type* (Section 2.4.4).

[6] Pierce's **case** [Pie91b] is a syntactic marker indicating where to apply the elimination rule. Clearly, a pure type assignment system should avoid this. We can avoid it even in a bidirectional system; see the rest of the thesis.

[7] If arguments were passed by name instead of by value, this would be unsound in a language with effects: evaluation of the same argument $e : A \vee B$ could sometimes return a value of type A and sometimes a value of type B. For example, let $e = e_1 \oplus e_2$ where $\oplus$ is a nondeterministic choice operator: both $e_1 \oplus e_2 \mapsto e_1$ and $e_1 \oplus e_2 \mapsto e_2$ are possible. Given $f(e)$ with $f = \lambda x. e'$, if x appears more than once in e', call by name evaluation would yield multiple copies of $e_1 \oplus e_2$,

$$\frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] : C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\Gamma \vdash \mathcal{E}[e'] : C}\ \vee E$$

The use of the evaluation context $\mathcal{E}$ guarantees that e' is the next expression to be evaluated (or is some value), following our informal reasoning above. In the example, $e' = g(x)$ and $\mathcal{E} = f\,[\,]$.

Several generalizations of this rule come to mind that are in fact unsound in our setting. For example, allowing simultaneous parallel case distinction over several occurrences of e', as in a rule proposed in [BDCd95],

$$\frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash e : C \\ \Gamma, x{:}B \vdash e : C\end{array}}{\Gamma \vdash [e'/x]\,e : C}\ \vee E'$$

is unsound here: two occurrences of the identical e' could return different results (the first of type A, the second of type B), while the rule above assumes consistency. Similarly, we cannot allow the occurrence of e' to be in a position where it might not be evaluated. That is, in $\vee E'$ it is not enough to require that there be exactly one occurrence of x in e, because, for example, if we consider the context

$$\begin{array}{lcl} f & : & ((1{\to}B) \to D) \wedge ((1{\to}C) \to D), \\ g & : & A \to (B \vee C), \\ x & : & A \end{array}$$

and term $f(\lambda y.\, g(x))$, then f may use its argument at multiple types, eventually evaluating $g(x)$ multiple times with different possible answers. Thus, treating it as if all occurrences must all have type B or all have type C is unsound. If we restrict the rule so that e' must be a value, as in [vBDCdM00], we obtain a sound but impractical rule—a typechecker would have to guess e' and, if it occurs more than once, a subset of its occurrences.

A final generalization suggests itself: we might allow the subterm e' to occur exactly once, and in any position where it would definitely have to be evaluated exactly once for the whole expression to be evaluated. Besides the difficulty of characterizing such positions, even this apparently innocuous generalization is unsound for the empty type $\bot$, as discussed in the next section.

2.4.2 The empty type

The 0-ary indefinite type is the empty or void type $\bot$; it has no values. For $\top$ we had one right subtyping rule; for $\bot$, following the principle of duality, we have one left rule:

$$\frac{}{\Gamma \vdash \bot \leq A}\ \bot L$$

For example, the term $\omega = (\mathbf{fix}\ u.\ \lambda x.\, u(x))()$ has type $\bot$. For an elimination rule $\bot E$, we can proceed by analogy with $\vee E$:

$$\frac{\Gamma \vdash e' : \bot \qquad \Gamma \vdash \mathcal{E}[e']\ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] : C}\ \bot E$$

As before, the expression typed must be an evaluation context $\mathcal{E}$ with redex e'. Viewing $\bot$ as a 0-ary union, we had two additional premises in $\vee E$, so we have none now. $\bot E$ is sound, but the generalization mentioned at the end of the previous section would violate progress (Theorem 2.21). This is easy to see through the counterexample $(()())\omega$.

2.4.3 Existential dependent types: Σ

Now we add an infinitary indefinite type Σ. Just as we have come to expect, the subtyping rules are dual to the rules for the corresponding definite type (in this case Π):

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B}\ \Sigma L \qquad \frac{\Gamma \vdash A \leq [i/b]\,B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B}\ \Sigma R$$

The typing rule that introduces Σ is simply

$$\frac{\Gamma \vdash e : [i/a]\,A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\, A}\ \Sigma I$$

For the elimination rule, we again have the restriction to evaluation contexts:

$$\frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \Sigma E$$

Not only is the restriction consistent with the elimination rules for $\bot$ and $\vee$, but it is required. The counterexample for $\bot$ suffices: Suppose that the rule were unrestricted, so that it typed any e containing some subterm e'. Let $e' = \omega$ and $e = (()())(\omega)$. Since $e' : \bot$, by subsumption e' has type $\Sigma a{:}\bot.\, A$ for any A, and by the contra rule, $a{:}\bot, x{:}A \vdash (()())x : C$ (where $\bot$ is the empty sort). Now we can apply the unrestricted rule to conclude $\vdash (()())e' : C$ for any C, contrary to progress.

2.4.4 Asserting types

The *asserting type* $P \mathbin{⅋} A$, read "P with A", is as the type A, but also asserts that P holds. To illustrate, suppose our index domain is the integers (index sort $\mathcal{Z}$) with propositions $P ::= i < j \mid i \leq j \mid \ldots$. Then

$$\mathbf{1} \to \Sigma a{:}\mathcal{Z}.\, ((a > 0) \mathbin{⅋} \mathsf{int}(a))$$

is the type of functions from unit to strictly positive integers.

On a high level, following Curry-Howard, we can think of a term of type $P \mathbin{⅋} A$ as a proof of both P and A, where only A has computational content. On an even higher level, we can think of $P \mathbin{⅋} A$ as simply another kind of conjunction that conjoins a proposition and a type, rather than two types as $\wedge$ does. Following this intuition, the subtyping rules and introduction rule are straightforward.

which would not all have to reduce in the same way. Suppose $e' = g\ x\ x$ where $g : (A \to A \to C) \wedge (B \to B \to C)$; then $f(e) \mapsto^* g\ (e_1 \oplus e_2)\ (e_1 \oplus e_2) \mapsto g\ e_2\ (e_1 \oplus e_2) \mapsto g\ e_2\ e_1$, which is ill typed.

Mutable references also serve: suppose $e = \mathtt{!r}$ where $r : (\mathsf{true} \vee \mathsf{false})$ **ref**. With the appropriate refinement of type bool, we can give e the type true $\vee$ false. However, occurrences of e will variously reduce to values of either type true or false, depending on intervening assignments to r.

$$\frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash (P \parr A) \leq B}\ \parr L \qquad \frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq (P \parr B)}\ \parr R$$

$$\frac{\Gamma \vdash e : A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : P \parr A}\ \parr I$$

The elimination rule, however, follows the pattern of ΣE.

$$\frac{\Gamma \vdash e' : P \parr A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \parr E$$

The subset sort mechanism, discussed above in the context of asserting types, can achieve the effect of $P \parr A$:

$$\mathbf{1} \to \Sigma a{:}\overbrace{\{a{:}\mathcal{Z} \mid a > 0\}}. \mathsf{int}(a) \qquad \textit{(Not in our system!)}$$

We consider $\parr$ to be an indefinite type. It may seem strange to have a type that looks like a conjunction in the same category as $\vee$ and Σ, which appear disjunctive. Consider that the *definite* types are closely linked to the allowed constructor types A^{con}: a constructor type is built up from intersections and guards of arrows $B \to \delta(i)$. The indefinite types, on the other hand, cannot be permitted in the codomain of a constructor type: if $\mathcal{S}(c) = B \to \bot$, we could construct values of the empty type! Likewise, if $\mathcal{S}(c) = B \to ((2 \doteq 3) \parr \delta(i))$, we can construct a value of type $(2 \doteq 3) \parr \delta(i)$, which asserts that $2 \doteq 3$. The fact that constructor types are as hostile to $\parr$ as to $\bot$ strongly suggests that $\parr$ is indefinite.[8]

The structure of the type system also suggests that $\parr$ belongs with $\vee$ and Σ. To take just one example, $\parr$L adds to the context in its premise, as ΣL does—and as ΠL does not.

2.4.5 Typechecking in evaluation order

The following rule internalizes a kind of substitution principle for evaluation contexts and allows us to check a term in evaluation order.

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \mathsf{direct}$$

Perhaps surprisingly, this rule is not only admissible but derivable in our system: from $e' : A$ we can conclude $e' : A \vee A$ and then apply $\vee$E. However, the corresponding bidirectional rule is not admissible, and so must be primitive in a bidirectional system (Chapter 3).

Thus, in either the type assignment or bidirectional systems, we can choose to typecheck the term in evaluation order. This has a clear parallel in Xi's work [Xi98], which is bidirectional and contains both Π and Σ. There, the order in which terms are typed is traditional, not guided by evaluation order. However, Xi's elaboration algorithm in the presence of Π and Σ transforms the term into a let-normal form, which has a similar effect. We return to this point in Chapter 5.

$$\text{Types } A, B, C, D ::= \mathbf{1} \mid A \to B \mid A * B \mid \delta(i) \mid A \wedge B \mid \top \mid \Pi a{:}\gamma.\, A \mid P \supset A \mid A \vee B \mid \bot \mid \Sigma a{:}\gamma.\, A \mid P \parr A$$

$\boxed{\Gamma \vdash e : A}$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma(u) = A}{\Gamma \vdash u : A}\ \mathsf{fixvar} \qquad \frac{\Gamma, u{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\, u.\, e : A}\ \mathsf{fix}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\ \mathsf{sub}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ {\to}I \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\ {\to}E$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 * A_2}\ *I \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{fst}(e) : A}\ *E_1 \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{snd}(e) : B}\ *E_2$$

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \qquad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\ \delta I \qquad \frac{\Gamma \vdash e : \delta(i) \qquad \Gamma \vdash ms :_{\delta(i)} C}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms : C}\ \delta E \qquad \frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e\ \mathsf{ok}}{\Gamma \vdash e : A}\ \mathsf{contra}$$

$$\frac{\Gamma \vdash v : A_1 \qquad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}\ \wedge I \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\ \wedge E_1 \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\ \wedge E_2$$

$$\frac{\Gamma \vdash v\ \mathsf{ok}}{\Gamma \vdash v : \top}\ \top I$$

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\ \Pi I \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\ \Pi E$$

$$\frac{\Gamma, P \vdash v : A}{\Gamma \vdash v : P \supset A}\ \supset I \qquad \frac{\Gamma \vdash e : P \supset A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : A}\ \supset E$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B}\ \vee I_1 \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B}\ \vee I_2 \qquad \frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] : C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\Gamma \vdash \mathcal{E}[e'] : C}\ \vee E$$

$$\frac{\Gamma \vdash e' : \bot \qquad \Gamma \vdash \mathcal{E}[e']\ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] : C}\ \bot E$$

$$\frac{\Gamma \vdash e : [i/a]\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\, A}\ \Sigma I \qquad \frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \Sigma E$$

$$\frac{\Gamma \vdash e : A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : P \parr A}\ \parr I \qquad \frac{\Gamma \vdash e' : P \parr A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \parr E$$

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \mathsf{direct}$$

Figure 2.9: Typing rules

$\boxed{\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e : C}$

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i' \vdash e : C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta' \qquad \Gamma, x{:}A \vdash e\ \mathsf{ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{F-ct}$$

$$\frac{\Gamma; c : A_1^{con}; c(x) : B \vdash e : C \quad \Gamma; c : A_2^{con}; c(x) : B \vdash e : C}{\Gamma; c : A_1^{con} \wedge A_2^{con}; c(x) : B \vdash e : C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : \Pi a{:}\gamma.\, A^{con}; c(x) : B \vdash e : C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : (P \supset A^{con}); c(x) : B \vdash e : C}\ \supset\text{-ct}$$

$\boxed{\Gamma \vdash ms :_B C}$

$$\frac{}{\Gamma \vdash \cdot :_B C}\ \mathsf{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e : C \qquad \Gamma \vdash ms :_B C}{\Gamma \vdash (c(x) \Rightarrow e \mid ms) :_B C}\ \mathsf{casearm}$$

Figure 2.10: Case typing rules

$\boxed{\Gamma \vdash A \leq B}$

$$\frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \to \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ \mathbf{1} \qquad \frac{\Gamma \vdash A_1 \leq B_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 * A_2 \leq B_1 * B_2}\ *$$

$$\frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ \wedge\mathrm{R} \qquad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ \wedge\mathrm{L}_1 \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ \wedge\mathrm{L}_2$$

$$\frac{\delta_1 \preceq \delta_2 \qquad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ \delta \qquad \frac{\Gamma \vdash [i/a]A \leq B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ \Pi\mathrm{L} \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ \Pi\mathrm{R}$$

$$\frac{\Gamma \vdash A_1 \leq B \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B}\ \vee\mathrm{L} \qquad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2}\ \vee\mathrm{R}_1 \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2}\ \vee\mathrm{R}_2$$

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B}\ \Sigma\mathrm{L} \qquad \frac{\Gamma \vdash A \leq [i/b]\, B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B}\ \Sigma\mathrm{R} \qquad \frac{}{\Gamma \vdash \bot \leq A}\ \bot\mathrm{L} \qquad \frac{}{\Gamma \vdash A \leq \top}\ \top\mathrm{R}$$

$$\frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash (P \supset A) \leq B}\ \supset\mathrm{L} \qquad \frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash A \leq (P \supset B)}\ \supset\mathrm{R}$$

$$\frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash (P \otimes A) \leq B}\ \otimes\mathrm{L} \qquad \frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq (P \otimes B)}\ \otimes\mathrm{R}$$

Figure 2.11: Subtyping rules

2.5 Properties of subtyping

We assume that $\models$ and $\vdash$ in the constraint domain are decidable. We also assume that we can somehow guess the index i in rules ΠL and ΣR (in practice, this requires introducing an existential variable and solving for it; see Chapter 6). Under these assumptions, we can show:

Theorem. $\Gamma \vdash A \leq B$ *is decidable.*

Proof. Straightforward, since for each subtyping rule (Figure 2.11), every premise is smaller than the conclusion. □

We omitted rules for reflexivity and transitivity of subtyping without loss of expressive power, because they are admissible:

Lemma 2.9 (Reflexivity and Transitivity of $\leq$)**.** *For any context* Γ*:*

(i) $\Gamma \vdash A \leq A$*;*

(ii) if $\Gamma \vdash A \leq B$ *and* $\Gamma \vdash B \leq C$ *then* $\Gamma \vdash A \leq C$*.*

Proof. For (i), by induction on A. For (ii), by induction on the given subtyping derivations; in each case at least one derivation becomes smaller. In the cases ΣR/ΣL and ΠR/ΠL we substitute an index i for a parameter a in a derivation. □

In addition we have a large set of inversion properties, which are purely syntactic in our system. We elide the lengthy statement of these properties here.

Note that we do not pretend to any completeness of subtyping with respect to the value interpretation (or anything else): the fact that the values of A are a subset of the values of B does not guarantee $A \leq B$. For example, both $\bot * \bot$ and $\bot$ are uninhabited, but $\bot * \bot \leq \bot$ cannot be derived.

2.6 Properties of values

For the proof of type safety, we need a key property: *values are always definite*. That is, once we obtain a value v, even though v might have type $A \vee B$, it *must* be possible to assign a definite type to v. In order to make this precise, we formulate substitutions σ that substitute values and indices, respectively, for several program variables x and index variables a. First we prove a simple lemma relating values and evaluation contexts.

Lemma 2.10. *If* $\mathcal{E}[e']$ value *then: (1)* e' value*; (2) for any* v value*,* $\mathcal{E}[v]$ value*.*

Proof. By structural induction on $\mathcal{E}$. □

$\boxed{\Gamma' \vdash \sigma : \Gamma}$

$$\frac{}{\Gamma' \vdash \cdot : \cdot}\ \text{empty-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \overline{\Gamma'} \models [\sigma]P}{\Gamma' \vdash \sigma : (\Gamma, P)}\ \text{prop-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \overline{\Gamma'} \vdash i : \gamma}{\Gamma' \vdash (\sigma, i/a) : (\Gamma, a{:}\gamma)}\ \text{ivar-}\sigma$$

$$\frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash v : [\sigma]A}{\Gamma' \vdash (\sigma, v/x) : (\Gamma, x{:}A)}\ \text{pvar-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash u : [\sigma]A}{\Gamma' \vdash \sigma : (\Gamma, u{:}A)}\ \text{fixvar-}\sigma$$

Figure 2.12: Substitution typing

2.6.1 Substitutions

Figure 2.12 defines a typing judgment for substitutions $\Gamma' \vdash \sigma : \Gamma$. It could be more general; here we are only interested in substitutions of values for program variables and indices for index variables that verify the logical assumptions of the constraint domain. Note in particular that substitutions σ do not substitute for fixed point variables, though rule fixvar-σ allows them to appear in the contexts; for example, one can derive $b{:}\mathcal{Z}, u{:}\mathsf{list}(b)\to\mathbf{1} \vdash b/a : (a{:}\mathcal{Z}, u{:}\mathsf{list}(b)\to\mathbf{1})$. Application of a substitution σ to a term e or type A is in the usual (capture-avoiding) manner.

Definition 2.11. The *identity substitution* Γ/Γ is defined thus:

$$\begin{aligned} \cdot/\cdot &= \cdot \\ (\Gamma, x{:}A)/(\Gamma, x{:}A) &= (\Gamma/\Gamma), x/x \\ (\Gamma, a{:}\gamma)/(\Gamma, a{:}\gamma) &= (\Gamma/\Gamma), a/a \\ (\Gamma, P)/(\Gamma, P) &= \Gamma/\Gamma \\ (\Gamma, u{:}A)/(\Gamma, u{:}A) &= \Gamma/\Gamma \end{aligned}$$

Proposition 2.12 (Identity Substitution Typing)**.** *For all Γ such that Γ/Γ is defined, $\Gamma, \Gamma' \vdash \Gamma/\Gamma : \Gamma$.*

Proof. By induction on Γ, using the easily proved identities $[\Gamma/\Gamma]\,A = A$, $[\Gamma/\Gamma]\,i = i$, $[\Gamma/\Gamma]\,P = P$. □

Lemma 2.13 (Weakening)**.**

(i) If $\Gamma \vdash B \le C$ then $\Gamma, x{:}A \vdash B \le C$.

(ii) If $\Gamma \vdash e : C$ then $\Gamma, x{:}A \vdash e : C$.

(iii) If $\Gamma' \vdash \sigma : \Gamma$ then $\Gamma', x{:}A \vdash \sigma : \Gamma$.

(And similarly for $\Gamma \vdash ms :_B C$ and $\Gamma; c : B^{con}; c(x) : \delta(i) \vdash e : C$.)

Proof. By induction on the given derivation. We show part (iii): For empty-σ the result is immediate; for ivar-σ and prop-σ, $\overline{\Gamma', x{:}A} = \overline{\Gamma'}$. For pvar-$\sigma$ and fixvar-σ, use the IH (ii). □

Lemma 2.14 (Substitution)**.** *(i) If $\Gamma \vdash A \le B$ and $\Gamma' \vdash \sigma : \Gamma$, then $\Gamma' \vdash [\sigma]A \le [\sigma]B$.*

(ii) If $\mathcal{D} :: \Gamma \vdash e : A$ and $\Gamma' \vdash \sigma : \Gamma$, then there exists $\mathcal{D}' :: \Gamma' \vdash [\sigma]e : [\sigma]A$. Moreover, if $\overline{\Gamma} = \Gamma$ (that is, Γ contains only index variables and index constraints), there is such a $\mathcal{D}'$ not larger than $\mathcal{D}$ (that is, there are no more typing rule applications in $\mathcal{D}'$ than in $\mathcal{D}$).

"$\mathcal{D}$ ::" is
"$\mathcal{D}$ deriv
(Sec. 1.9

(iii) If $\Gamma \vdash e' : B$ and $\Gamma, u{:}B \vdash e : A$ then $\Gamma \vdash [e'/u]\,e : A$.

(And similar properties hold for $\Gamma \vdash ms :_C B$ and $\Gamma; c : C^{con}; c(x) : \delta(i) \vdash e : A$.)

Proof. By induction on the respective derivations. □

2.6.2 Definiteness

A typing judgment is *definite* if, whenever it shows that a term has indefinite type, we can also show that the term has a more particular type. For example, the judgment $\cdot \vdash \mathsf{Cons}(0, \mathsf{Nil}) : \mathsf{list}(0) \vee \mathsf{list}(1)$ is definite because we can also show $\cdot \vdash \mathsf{Cons}(0, \mathsf{Nil}) : \mathsf{list}(1)$. In this section, we show that every judgment typing a closed value is definite. More specifically, we show that

(i) $\vdash v : \bot$ is *not* derivable;

(ii) if $\vdash v : A \vee B$ then $\vdash v : A$ or $\vdash v : B$;

(iii) if $\vdash v : \Sigma a{:}\gamma.\, A$ then $\vdash v : [i/a]A$ for some i;

(iv) if $\vdash v : P \otimes A$ then $\vdash v : A$ and $\models P$.

For the proof of definiteness (and the later proofs of value inversion and type safety), we use a nontrivial measure of derivation size. This measure is motivated by the premises of the indefinite elimination rules such as $\vee$E, where we must substitute for a variable in the proof cases for the indefinite elimination rules, such as $\vee$E. That rule includes a premise introducing a new variable x. We will have a value e' to substitute for x, but if we use only a simple measure of derivation size, we would be unable to apply the IH to the resulting derivation (which is required) because it could become larger. We obtain a valid inductive proof by stratifying derivations into those of *rank* 0, 1, etc., where the rank of $\mathcal{D}$ is (roughly) the number of applications of indefinite elimination rules in $\mathcal{D}$. We will ensure that the derivations $\mathcal{D}'$ *produced* by the theorem have rank 0. Thus, applying the IH to the typing for the value e', and substituting e' for x, cannot increase rank because the substituted derivations typing e' have rank 0—and the rank of a subderivation of any premise of $\vee$E is one less than the rank of the original derivation, because that original derivation ends in $\vee$E.

Of course, we also want to apply the IH to premises of other rules, say $\wedge$I, where the rank of the derivation of the premises is no smaller, so we use a lexicographical ordering of the rank (notated *Rank*) and the usual measure of derivation size. We also define the rank of any derivation concluding in certain rules to be 0, regardless of applications of $\vee$E, etc. in its subderivations. This lets us handle cases like $e' = \lambda x.\, e_1$ where the typing of e_1 itself uses $\vee$E. Substituting a typing derivation of e' into another derivation $\mathcal{D}_1$ may increase the number of times $\vee$E is used in $\mathcal{D}_1$,

[8] Σ with an empty index sort $\bot$ can also construct an abomination, if $\mathcal{S}(c) = B \to \Sigma a{:}\bot.\, \delta(a)$. Allowing unions in constructor codomains is not immediately catastrophic, but would break our key property of value definiteness (below).

for example, if $\mathcal{D}_1$ uses $\wedge$I. But we can ignore those uses since they are "buried" inside the typing derivation of e_1, and the proof never penetrates the derivation concluding in $\to$I.

An illustration of this notion is given in Figure 2.13. Rank depends only on the rules applied and not the judgments, so we elide the judgments, showing ranks in their place. At the root of the derivation, direct is applied to subderivations of ranks 0 and 1, resulting in a derivation of rank 2. The right-hand subderivation concludes in an application of $\wedge$I; its rank is just the sum of its subderivations' ranks ($1 + 0 = 1$). Moving up, we have a $\vee$E application to three rank 0 derivations, yielding a rank 1 derivation. However, the ΣE application at the upper right, which yields a derivation of rank 1, does not add to the rank of the larger derivation because there is an intervening $\to$I, forcing the rank to be 0.

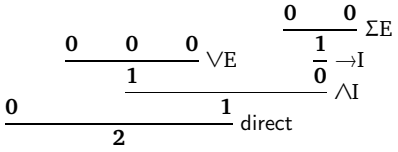

Figure 2.13: Illustration of derivation rank

Definition 2.15 (Definiteness Derivation Measure)**.** The measure μ of a derivation $\mathcal{D}$ is a lexicographically ordered pair

$$\mu(\mathcal{D}) = \langle \mathit{Rank}(\mathcal{D}), \mathit{Size}(\mathcal{D}) \rangle$$

Let $\mathcal{R}$ be the rule concluding $\mathcal{D}$, and let $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be the n subderivations, one for each premise of $\mathcal{R}$ deriving a typing judgment. Then $\mathit{Size}(\mathcal{D})$ is just the simple derivation size, that is,

$$\mathit{Size}(\mathcal{D}) = 1 + \mathit{Size}(\mathcal{D}_1) + \cdots + \mathit{Size}(\mathcal{D}_n)$$

and $\mathit{Rank}(\mathcal{D})$ is defined as follows.

$$\mathit{Rank}(\mathcal{D}) = \begin{cases} 0 & \text{if } \mathcal{R} \text{ is fix or } \to\text{I} \\ 1 + \mathit{Rank}(\mathcal{D}_1) + \cdots + \mathit{Rank}(\mathcal{D}_n) & \text{if } \mathcal{R} \text{ is } \bot\text{E}, \vee\text{E}, \Sigma\text{E}, \parr\text{E} \text{ or direct} \\ \mathit{Rank}(\mathcal{D}_1) + \cdots + \mathit{Rank}(\mathcal{D}_n) & \text{otherwise} \end{cases}$$

Section 1.9.2 explains some of the notation used in our rules.

We also define $\mathit{Rank}(\sigma)$ of a substitution $\Gamma' \vdash \sigma : \Gamma$ as the sum of the ranks of its constituent typing derivations, slightly abusing terminology since it is really the *derivation* of the substitution typing judgment that has rank. Thus, if $\sigma = e'/x$ has rank 0, the derivation of $\Gamma' \vdash e' : A$ must also have rank 0.

We also need a "ranked" version of the substitution lemma (2.14) that says, roughly, that substituting rank 0 derivations for rank 0 derivations (specifically, applications of var) cannot increase rank; therefore, applying a rank 0 substitution does not increase the rank of the resulting typing derivation. For consistency with the earlier lemma, we designate the whole thing "(ii)".

Lemma 2.16 (Ranked Substitution)**.**

(ii) If $\mathcal{D} :: \Gamma \vdash e : A$ *and* $\cdot \vdash \sigma : \Gamma$*, where* $\mathit{Rank}(\sigma) = 0$ *then there exists* $\mathcal{D}' :: \cdot \vdash [\sigma]e : [\sigma]A$*, where* $\mathit{Rank}(\mathcal{D}') = \mathit{Rank}(\mathcal{D})$*.*

Moreover, if $\overline{\Gamma} = \Gamma$ *it is the case that* $\mathit{Size}(\mathcal{D}') \leq \mathit{Size}(\mathcal{D})$*.*

Proof. By induction on the derivation $\mathcal{D}$.

In the fix and $\to$I cases, we apply Lemma 2.14 on each premise, then apply the same rule, yielding a derivation that is rank 0 (according to Definition 2.15). Since $\mathit{Rank}(\mathcal{D})$ is also 0, we have the result.

In the var case, e is some variable x. Within the derivation of $\cdot \vdash \sigma : \Gamma$ there must lie a derivation $\mathcal{D}' :: \cdot \vdash e' : A$, where $[\sigma]x = e'$ and $\Gamma(x) = A$. We have $\mathit{Rank}(\sigma) = 0$, so $\mathit{Rank}(\mathcal{D}') = 0$, which was to be shown.

In all other cases, we simply apply the IH to each premise and apply the same rule. (For the rules that themselves add to the rank, such as $\vee$E, we have a derivation of rank n whose subderivations have ranks summing to $n - 1$. Applying the IH to each yields a derivation of the same rank. Therefore, applying $\vee$E to those derivations results in a derivation of rank $(n - 1) + 1 = n$.) For rules in which some premises are not typing judgments, such as sub and δE, we use Lemma 2.14 on them.

The "moreover" part follows from the "moreover" part of Lemma 2.14. □

Theorem 2.17 (Value Definiteness)**.** *If* $\mathcal{D} :: \cdot \vdash v : A$ *then:*

(i) $\cdot \vdash A \leq \bot$ *is* not *derivable;*

(ii) if $\cdot \vdash A \leq B_1 \vee B_2$ *then there exists* $\mathcal{D}' :: \cdot \vdash v : B_k$ *for some* $k \in \{1, 2\}$*;*

(iii) if $\cdot \vdash A \leq \Sigma b{:}\gamma.\, B$ *then there exists* $\vdash i : \gamma$ *such that* $\mathcal{D}' :: \cdot \vdash A' \leq [i/b]B$*;*

(iv) if $\cdot \vdash A \leq P \parr B$ *then* $\models P$ *and* $\mathcal{D}' :: \cdot \vdash v : B$*;*

(v) $\mathcal{D}' :: \cdot \vdash v : A$.[9]

Moreover, in parts (ii)–(v), $\mathit{Rank}(\mathcal{D}') = 0$*.*

Proof. By induction on $\mu(\mathcal{D})$, where $\mathcal{D} :: \cdot \vdash v : A$.

The term v is a value, so we need not consider rules that cannot type values. Furthermore, by Property 2.5 the contra case cannot arise. Most cases follow easily from the IH, reflexivity/-transitivity of subtyping, and rule sub. For example, in the case for ΠI, part (ii), we are given $\cdot \vdash \Pi a{:}\gamma.\, A_0 \leq B_1 \vee B_2$. The only rules that can derive such a judgment are ΠL, $\vee R_1$ and $\vee R_2$. If ΠL was used, use Lemma 2.16 to eliminate a in the premise of ΠI and apply the IH. If $\vee R_1$ was used, we have $\cdot \vdash \Pi a{:}\gamma.\, A_0 \leq B_1$, whence we can apply the IH. The $\vee R_2$ subcase is similar.

Rule var is impossible since the context is empty.

For sub, use transitivity of subtyping followed by the IH.

That leaves the contextual rules $\bot$E, $\vee$E, ΣE, $\parr$E and direct; we show the first three cases (the last two are similar to $\vee$E, but using IH(iv) and IH(v), respectively):

[9] The force of part (v) comes from the "Moreover, ..." statement about $\mathit{Rank}(\mathcal{D}')$.

- **Case** $\bot$E : $\mathcal{D} :: \dfrac{\cdot \vdash e' : \bot \qquad \cdot \vdash \mathcal{E}[e']\ \mathsf{ok}}{\cdot \vdash \mathcal{E}[e'] : A}$

  By assumption, $\mathcal{E}[e']$ is a value. By Lemma 2.10, e' is a value. $\cdot \vdash \bot \leq \bot$ (Lemma 2.9), so by the IH (i), this case cannot arise.

- **Case** $\wedge$I : $\mathcal{D} :: \dfrac{\cdot \vdash v : A_1 \qquad \cdot \vdash v : A_2}{\cdot \vdash v : A_1 \wedge A_2}$

  This style of line-by-line proof is explained in section 9.2.

  Part (ii): It is given that $\vdash A_1 \wedge A_2 \leq B_1 \vee B_2$. The only rules that can derive such a judgment are $\vee R_{1,2}$ and $\wedge L_{1,2}$.

  If by $\vee R_1$, we have $\vdash A_1 \wedge A_2 \leq B_1$. Applying the IH (v) to each subderivation of $\mathcal{D}$ yields rank-0 derivations of $\vdash v : A_1$ and $\vdash v : A_2$; applying $\wedge$I to those yields a rank-0 derivation of $\vdash v : A_1 \wedge A_2$. By sub, $\vdash v : B_1$, which was to be shown. The $\vee R_2$ case is similar.

  If by $\wedge L_1$,we have $\vdash A_1 \leq B_1 \vee B_2$; the result follows by IH on $\cdot \vdash v : A_1$. The $\wedge L_2$ case is similar.

  The proofs of parts (i) and (iii)–(v) are similar.

- **Case** ΠI, $\supset$I, $\top$I : Similar to $\wedge$I.

- **Case** $\wedge E_1$: $\mathcal{D} :: \dfrac{\cdot \vdash v : A \wedge A_0}{\cdot \vdash v : A}$

  Part (ii): We have $\vdash A \leq B_1 \vee B_2$. By $\wedge L_1$ and transitivity (Lemma 2.9), $\cdot \vdash A \wedge A_0 \leq B_1 \vee B_2$. The result follows by IH on $\cdot \vdash v : A \wedge A_0$.

  The proofs of parts (i) and (iii)–(v) are similar.

- **Case** $\wedge E_2$, ΠE, $\supset$E : Similar to $\wedge E_1$.

- **Case** $\vee$E : $\mathcal{D} :: \dfrac{\cdot \vdash e' : C_1 \vee C_2 \qquad \overset{\mathcal{D}_1}{x{:}C_1 \vdash \mathcal{E}[x] : A} \qquad \overset{\mathcal{D}_2}{y{:}C_2 \vdash \mathcal{E}[y] : A}}{\cdot \vdash \mathcal{E}[e'] : A}$

  This case goes the same way for all parts, (i)–(v).

  $\mathcal{E}[e']$ value is given. By Lemma 2.10, $\mathcal{E}[x]$ value and e' value.

  By Lemma 2.9, $\vdash C_1 \vee C_2 \leq C_1 \vee C_2$. By IH (ii), $\mathcal{D}' :: \cdot \vdash e' : C_k$ for some $k \in \{1,2\}$, where $Rank(\mathcal{D}') = 0$. Assume $k = 1$; the $k = 2$ case is similar. By empty-σ and pvar-σ, $\cdot \vdash e'/x : x{:}C_1$, which, since $Rank(\mathcal{D}') = 0$, is a rank-0 substitution. By Lemma 2.16 on $\mathcal{D}_1 :: x{:}C_1 \vdash \mathcal{E}[x] : A$, we have

  $$\mathcal{D}'' :: \cdot \vdash [e'/x]\ \mathcal{E}[x] : [e'/x]A$$

  where $Rank(\mathcal{D}'') = Rank(\mathcal{D}_1)$. Since x is new, it does not appear in $\mathcal{E}$, so in fact we have $\mathcal{D}'' :: \cdot \vdash \mathcal{E}[e'] : A$. By Definition 2.15, $Rank(\mathcal{D}_1) < Rank(\mathcal{D})$. Therefore $Rank(\mathcal{D}'') < Rank(\mathcal{D})$.

Under the lexicographic ordering, $\mu(\mathcal{D}'') < \mu(\mathcal{D})$, so we can apply the IH to $\mathcal{D}'' :: \cdot \vdash \mathcal{E}[e'] : A$, yielding the result.

- **Case** direct : $\mathcal{D} :: \dfrac{\cdot \vdash e' : C \qquad \overset{\mathcal{D}_1}{x{:}C \vdash \mathcal{E}[x] : A}}{\cdot \vdash \mathcal{E}[e'] : A}$

$\cdot \vdash e' : C$	Subderivation
$\mathcal{D}'_0 :: \cdot \vdash e' : C$ and $Rank(\mathcal{D}'_0) = 0$	By IH (v)
$\cdot \vdash e'/x : x{:}C$	By empty-σ then pvar-σ
$\mathcal{D}_1 :: x{:}C \vdash \mathcal{E}[x] : A$	Subderivation
$\mathcal{D}' :: \cdot \vdash [e'/x]\ \mathcal{E}[x] : A$	By Lemma 2.16
$Rank(\mathcal{D}') = Rank(\mathcal{D}_1)$	By Lemma 2.16
$Rank(\mathcal{D}_1) < Rank(\mathcal{D})$	By Definition 2.15
$[e'/x]\ \mathcal{E}[x]$ satisfies (i)–(v)	By IH
$[e'/x]\ \mathcal{E}[x] = \mathcal{E}[e']$	By defn. of substitution (x new)
☞ $\mathcal{E}[e']$ satisfies (i)–(v)	By previous equation

- **Case** $\otimes$E : $\mathcal{D} :: \dfrac{\cdot \vdash e' : P \otimes C \qquad \overset{\mathcal{D}_1}{P, x{:}C \vdash \mathcal{E}[x] : A}}{\cdot \vdash \mathcal{E}[e'] : A}$

$\cdot \vdash e' : P \otimes C$	Subderivation
$\models P$	By IH (iv)
$\mathcal{D}'_0 :: \cdot \vdash e' : C$ and $Rank(\mathcal{D}'_0) = 0$	By IH (iv)
$\vdash e'/x : x{:}C$	By empty-σ then pvar-σ
$\vdash e'/x : x{:}C, P$	By prop-σ
$\mathcal{D}_1 :: x{:}C, P \vdash \mathcal{E}[x] : A$	Subderivation
$\mathcal{D}' :: \cdot \vdash [e'/x]\ \mathcal{E}[x] : A$	By Lemma 2.16
$Rank(\mathcal{D}') = Rank(\mathcal{D}_1)$	By Lemma 2.16
$Rank(\mathcal{D}_1) < Rank(\mathcal{D})$	By Definition 2.15
$[e'/x]\ \mathcal{E}[x]$ satisfies (i)–(v)	By IH
$[e'/x]\ \mathcal{E}[x] = \mathcal{E}[e']$	By defn. of substitution (x new)
☞ $\mathcal{E}[e']$ satisfies (i)–(v)	By previous equation

- **Case** ΣE : Similar to the $\otimes$E case, using IH (iii) instead of IH (iv), and the substitution $i/a, e'/x$. □

2.6.3 Value inversion on $\to$, $*$, $\delta(\mathrm{i})$

For ordinary types (not property types) we have a value inversion lemma (also known as *genericity* or *canonical forms*) used in the respective elimination rule cases in the type safety proof (Thm. 2.21): for instance, that proof's case for $*\mathrm{E}_1$ must invert $v : \mathrm{A}_1 * \mathrm{A}_2$ to obtain $v = (v_1, v_2)$ where $v_1 : \mathrm{A}_1$. (The unit type **1** has no elimination rule, so it does not need an inversion lemma.)[10]

To get through the value inversion proof cases for the indefinite type elimination rules, such as $\vee$E, we use the same derivation measure μ that we just used to prove value definiteness. However, the statement of the proposition is simpler because, in those cases, we use value definiteness—not the IH—on the subterm e'. Thus we need not say the resulting derivation $\mathcal{D}'$ is smaller than $\mathcal{D}$; we do not even name the resulting derivation.

Lemma 2.18 (Value Inversion for $\to$, $*$, $\delta(\mathrm{i})$)**.** *If* $\mathcal{D} :: \cdot \vdash v : \mathrm{A}$*, then*

(i) if $\cdot \vdash \mathrm{A} \leq \mathrm{B}_1 \to \mathrm{B}_2$ *then* $v = \lambda x.\, e$ *where* $\cdot \vdash [v'/x]\, e : \mathrm{B}_2$ *for any* $\vdash v' : \mathrm{B}_1$*;*

(ii) if $\cdot \vdash \mathrm{A} \leq \mathrm{B}_1 * \mathrm{B}_2$ *then* $v = (v_1, v_2)$ *where* $\cdot \vdash v_1 : \mathrm{B}_1$ *and* $\cdot \vdash v_2 : \mathrm{B}_2$*.*

(iii) if $\cdot \vdash \mathrm{A} \leq \delta(\mathrm{i})$ *then* $v = \mathrm{c}(v')$ *where* $\cdot \vdash \mathcal{S}(\mathrm{c}) \uparrow \mathrm{B}_1 \to \delta'(\mathrm{i}')$ *and* $\cdot \vdash \mathrm{c}(v') : \delta'(\mathrm{i}')$ *and* $\cdot \vdash v' : \mathrm{B}_1$ *and* $\cdot \vdash \delta'(\mathrm{i}') \leq \delta(\mathrm{i})$*.*

Proof. By induction on the measure $\mu(\mathcal{D})$, where $\mathcal{D} :: \cdot \vdash v : \mathrm{A}$. We show part (i), inversion on $\to$; for part (ii), inversion on $*$, all cases except $\to$I and $*$I are similar to the corresponding cases for part (i), while the $*$I case for (ii) is a slight simplification of the $\to$I case for (i). For part (iii), $\delta(\mathrm{i})$, all cases are similar to part (i) except $\to$I and δI; we show the δI case in full at the end.

Rules fixvar, fix, $\to$E, $*\mathrm{E}_1$, $*\mathrm{E}_2$, and δE cannot type values. For **1**I there is no way to derive $\cdot \vdash \mathbf{1} \leq \mathrm{B}_1 \to \mathrm{B}_2$, so the rule could not have been used; $*$I, δI, and $\top$I are similar.

Since the context is empty, the var case is impossible.

For sub, $\wedge\mathrm{E}_{1,2}$, ΠE, $\supset$E, $\vee\mathrm{I}_{1,2}$, ΣI, and $\otimes$I, we show that the type in each rule's premise is a subtype of A (immediate in sub, otherwise by the appropriate subtyping rule), then use transitivity of subtyping and apply the IH.

For $\bot$E, $\vee$E, ΣE, $\otimes$E and direct, in which the subject term is $\mathcal{E}[v']$, we use Theorem 2.17, yielding $\mathcal{D}' :: \cdot \vdash v' : \mathrm{C}$ (for appropriate C) where $\mathit{Rank}(\mathcal{D}') = 0$. By an argument similar to that in the $\vee$E case of the proof of Theorem 2.17, we obtain a derivation $\mathcal{D}'_1 :: \cdot \vdash \mathcal{E}[v'] : \mathrm{A}$ such that $\mathit{Rank}(\mathcal{D}'_1) < \mathit{Rank}(\mathcal{D})$, and therefore $\mu(\mathcal{D}'_1) < \mu(\mathcal{D})$, so we can apply the IH to $\mathcal{D}'_1$. We show the ΣE case below.

The contra case is excluded by Property 2.5.

The remaining cases are $\to$I, ΠI, $\wedge$I and $\supset$I.

- **Case** $\to$I : $\mathcal{D} :: \dfrac{\mathrm{x}{:}\mathrm{A}_1 \vdash e : \mathrm{A}_2}{\cdot \vdash \lambda x.\, e : \mathrm{A}_1 \to \mathrm{A}_2}$

[10] The combination of the inversion properties for $\to$, $*$, and $\mathrm{d}(\mathrm{i})$ into one lemma (Lemma 2.18) is not required; there is no interplay between these constructors in the type system, in particular, in the subtyping rules.

The result that $v = \lambda x.\, e$ is immediate. We have $\cdot \vdash \mathrm{A} \leq \mathrm{B}_1 \to \mathrm{B}_2$ and $\mathrm{A} = \mathrm{A}_1 \to \mathrm{A}_2$. By inversion, $\cdot \vdash \mathrm{B}_1 \leq \mathrm{A}_1$ and $\cdot \vdash \mathrm{A}_2 \leq \mathrm{B}_2$. For all v' such that $\cdot \vdash v' : \mathrm{B}_1$:

	$\cdot \vdash \mathrm{B}_1 \leq \mathrm{A}_1$	Above
	$\cdot \vdash v' : \mathrm{B}_1$	Assumption
	$\cdot \vdash v' : \mathrm{A}_1$	By sub
	$\cdot \vdash v'/x : x{:}\mathrm{A}_1$	By empty-σ and pvar-σ
	$x{:}\mathrm{A}_1 \vdash e : \mathrm{A}_2$	Subd.
	$\cdot \vdash [v'/x]e : [v'/x]\mathrm{A}_2$	By Lemma 2.14
	$\cdot \vdash [v'/x]e : \mathrm{A}_2$	$x \notin \mathit{FV}(\mathrm{A}_2)$
	$\cdot \vdash \mathrm{A}_2 \leq \mathrm{B}_2$	Above
☞	$\cdot \vdash [v'/x]e : \mathrm{B}_2$	By sub

- **Case** ΠI : $\mathcal{D} :: \dfrac{a{:}\gamma \vdash v : \mathrm{A}'}{\cdot \vdash v : \Pi a{:}\gamma.\, \mathrm{A}'}$

We have $\mathrm{A} = \Pi a{:}\gamma.\, \mathrm{A}'$.

	$\cdot \vdash \Pi a{:}\gamma.\, \mathrm{A}' \leq \mathrm{B}_1 \to \mathrm{B}_2$	Given
	$\cdot \vdash [i/a]\mathrm{A}' \leq \mathrm{B}_1 \to \mathrm{B}_2$ and $\cdot \vdash i : \gamma$	By inversion (only rule ΠL possible)
	$\cdot \vdash i/a : a{:}\gamma$	By empty-σ and ivar-σ
	$\mathcal{D}' :: a{:}\gamma \vdash v : \mathrm{A}'$	Subd.
	$\mathcal{D}'' :: \cdot \vdash [i/a]v : [i/a]\mathrm{A}'$ and $\mathit{Rank}(\mathcal{D}'') = \mathit{Rank}(\mathcal{D}')$ and $\mathit{Size}(\mathcal{D}'') \leq \mathit{Size}(\mathcal{D}')$	By Lemma 2.16
	$\mu(\mathcal{D}'') \leq \mu(\mathcal{D}')$	By Definition 2.15
	$\mathcal{D}'' :: \cdot \vdash v : [i/a]\mathrm{A}'$	a not free in v
☞	$v = \lambda x.\, e$ and $\cdot \vdash [v'/x]\, e : \mathrm{B}_2$ for any v' s.t. $\cdot \vdash v' : \mathrm{B}_1$	By IH

- **Case** $\supset$I : Similar to the ΠI case.

- **Case** $\wedge$I : $\mathcal{D} :: \dfrac{\cdot \vdash v : \mathrm{A}_1 \quad \cdot \vdash v : \mathrm{A}_2}{\cdot \vdash v : \mathrm{A}_1 \wedge \mathrm{A}_2}$

We have $\mathrm{A} = \mathrm{A}_1 \wedge \mathrm{A}_2$, and $\cdot \vdash \mathrm{A} \leq \mathrm{B}_1 \to \mathrm{B}_2$. The only subtyping rules that could have been used are $\wedge\mathrm{L}_{1,2}$. These cases are symmetric; assume $\wedge\mathrm{L}_2$. The premise of $\wedge\mathrm{L}_2$ is $\cdot \vdash \mathrm{A}_2 \leq \mathrm{B}_1 \to \mathrm{B}_2$. Applying the IH to $\cdot \vdash v : \mathrm{A}_2$ yields $v = \lambda x.\, e$ where $\vdash [v'/x]\, e : \mathrm{B}_2$ for any v' such that $\vdash v' : \mathrm{B}_1$, which was to be shown.

- **Case** ΣE : $\mathcal{D} :: \dfrac{\cdot \vdash e' : \Sigma a{:}\gamma.\, C \qquad a{:}\gamma, x{:}C \vdash \mathcal{E}[x] : A}{\cdot \vdash \mathcal{E}[e'] : A}$

  By Theorem 2.17, there exists $\vdash i : \gamma$ such that $\cdot \vdash e' : [i/a]C$ by a rank-0 derivation. By Lemma 2.16, $\vdash [i/a, e'/x]\ \mathcal{E}[e'] : A$ by a derivation of the same rank as the subderivation $a{:}\gamma, x{:}C \vdash \mathcal{E}[x] : A$. That rank is one less than the rank of $\mathcal{D}$, so we can apply the IH, yielding the result.

That concludes parts (i) and (ii). For part (iii), we show the δI case:

- **Case** δI : $\mathcal{D} :: \dfrac{\cdot \vdash c : B_1 \to \delta'(i') \qquad \cdot \vdash v' : B_1}{\cdot \vdash c(v') : \delta'(i')}$

  The only rule that can conclude $\cdot \vdash c : B_1 \to \delta'(i')$ is $\mathcal{S}$-con, which has premise

☞	$\cdot \vdash \mathcal{S}(c) \uparrow B_1 \to \delta'(i')$	
	$A = \delta'(i')$	Given
☞	$\cdot \vdash c(v') : \delta'(i')$	Conclusion of derivation
☞	$v = c(v')$	Given
	$\cdot \vdash A \le \delta(i)$	Given
☞	$\cdot \vdash v' : B_1$	Subderivation
☞	$\cdot \vdash \delta'(i') \le \delta(i)$	By $A = \delta'(i')$

  □

2.6.4 Lemmas for case

We prove two lemmas relating to constructors and matches:

- Lemma 2.19 (used in Lemma 2.20) allows inversion on judgments of the form $\cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$, obtaining $x{:}A_1 \vdash e : C$ where A_1 is appropriately related to A^{con};
- Lemma 2.20 shows that given a value $c(v) : \delta(i)$ and a typing of $ms = \ldots c(x) \Rightarrow e \ldots$, there is an A such that $v : A$ and $x{:}A \vdash e : C$, pursuant to showing that **case** $c(v)$ **of** $ms \mapsto_R [v/x]e$ preserves typing (in the δE case of Theorem 2.21).

Lemma 2.19. *If* $\cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$ *where* $\cdot \vdash A^{con} \uparrow A_1 \to \delta'(i')$ *and* $\delta' \preceq \delta$ *and* $\cdot \models i' \doteq i$ *then* $x{:}A_1 \vdash e : C$.

Proof. By induction on the size of $\mathcal{D} :: \cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$.

- **Case** δS-ct : $\mathcal{D} :: \dfrac{\delta'' \preceq \delta \qquad x{:}A', i'' \doteq i \vdash e : C}{\cdot; c : A' \to \delta''(i''); c(x) : \delta(i) \vdash e : C}$

  Only rule refl-$\uparrow$ could have derived $\cdot \vdash A' \to \delta''(i'') \uparrow A_1 \to \delta'(i')$, so we have $A' = A_1$ and $\delta'' = \delta'$ and $i'' = i'$. The second subderivation is therefore $x{:}A_1, i' \doteq i \vdash e : C$. It is given that $\cdot \models i' \doteq i$. By Lemma 2.14 with substitution x/x we obtain $x{:}A_1 \vdash e : C$, which was to be shown.

- **Case** δF-ct : $\mathcal{D} :: \dfrac{\delta'' \not\preceq \delta \qquad x{:}A \vdash e \text{ ok}}{\cdot; c : A' \to \delta''(i); c(x) : \delta(i) \vdash e : C}$

  We have the premise $\delta'' \not\preceq \delta$. By analogy with the preceding case, $\delta'' = \delta'$, so $\delta' \not\preceq \delta$. This contradicts $\delta' \preceq \delta$, which was given: the case is impossible.

- **Case** $\wedge$-ct : $\mathcal{D} :: \dfrac{\cdot; c : A^{con\prime}_1; c(x) : \delta(i) \vdash e : C \qquad \cdot; c : A^{con\prime}_2; c(x) : \delta(i) \vdash e : C}{\cdot; c : A^{con\prime}_1 \wedge A^{con\prime}_2; c(x) : \delta(i) \vdash e : C}$

  Inversion on $\cdot \vdash A^{con\prime}_1 \wedge A^{con\prime}_2 \uparrow A_1 \to \delta'(i')$ yields $\cdot \vdash A^{con\prime}_k \uparrow A_1 \to \delta'(i')$ for some $k \in 1..2$. The result follows by IH on the kth premise.

- **Case** Π-ct, $\supset$-ct : Use inversion on $\cdot \vdash A^{con} \uparrow A_1 \to \delta'(i')$ to get $\cdot \vdash i : \gamma$ or $\cdot \models P$, and construct a substitution i/a or $\cdot$. Use Lemma 2.14 on a subderivation. Since $\overline{a{:}\gamma} = a{:}\gamma$ (and $\overline{\cdot} = \cdot$), the resulting derivation is no larger than the subderivation. The IH then gives the result. □

Lemma 2.20. *If* $\cdot \vdash c(v) : \delta(i)$ *and* $\cdot \vdash ms :_{\delta(i)} C$ *and* $ms = \ldots c(x) \Rightarrow e \ldots$ *then there exists* A *such that* $x{:}A \vdash e : C$ *where* $\cdot \vdash v : A$.

Proof. By induction on the derivation of $\cdot \vdash ms :_{\delta(i)} C$. Two rules can derive such judgments:

- **Case** emptyms : $ms = \cdot$, but it is given that ms includes $c(x) \Rightarrow e$. This rule could not have been used.

- **Case** casearm : $\mathcal{D} :: \dfrac{\cdot; c' : \mathcal{S}(c'); c'(x') : \delta(i) \vdash e' : C \qquad \cdot \vdash ms' :_{\delta(i)} C}{\cdot \vdash (c'(x') \Rightarrow e') \mid ms' :_{\delta(i)} C}$

  If $c' \neq c$: It is given that $ms = (c'(x') \Rightarrow e') \mid ms'$ includes $c(x) \Rightarrow e$. Since $c' \neq c$, ms' must include $c(x) \Rightarrow e$. The result follows by IH on $\cdot \vdash ms' :_{\delta(i)} C$.

  If $c' = c$: By the assumption that each constructor appears in exactly one case arm, $c'(x') \Rightarrow e' = c(x) \Rightarrow e$. Therefore the first subderivation is

  $$\cdot; c : \mathcal{S}(c); c(x) : \delta(i) \vdash e : C.$$

  $\cdot \vdash \delta(i) \le \delta(i)$ by Lemma 2.9. By Property 2.5, $\cdot \not\models \bot$. Thus we can use Lemma 2.18 on $\cdot \vdash c(v) : \delta(i)$ to yield A and $\delta'(i')$ such that $\cdot \vdash \mathcal{S}(c) \uparrow A \to \delta'(i')$ where $\cdot \vdash v : A$ and $\cdot \vdash \delta'(i') \le \delta(i)$. By inversion on the latter, $\delta' \preceq \delta$ and $\cdot \models i' \doteq i$.

  We can now apply Lemma 2.19 to show $x{:}A \vdash e : C$, which with $\cdot \vdash v : A$ constitutes the result. □

2.7 Type preservation and progress

Having proved value definiteness, value inversion, and properties of case expressions, we are ready to prove type safety. We prove the preservation and progress theorems simultaneously; we could

prove them separately, but the proofs would share so much structure as to be more cumbersome than the simultaneous proof.[11] Note that while we have elided any explicit effects from the present system, the analysis in [DP00] applies in this setting. Again we use the derivation measure $\mu(\mathcal{D})$, which allows application of the IH after substitution in a subderivation in the direct, $\bot$E, $\vee$E, ΣE and $\otimes$E cases.

Theorem 2.21 (Type Preservation and Progress)**.** *If* $\mathcal{D} :: \cdot \vdash e : C$ *then either*

(1) e value*, or*

(2) there exists e' *such that* $e \mapsto e'$ *and* $\vdash e' : C$.

Proof. By induction on the measure $\mu(\mathcal{D})$ of the derivation $\mathcal{D} :: \cdot \vdash e : C$. If e value, the result is immediate. So suppose e is not a value.

Rules **1**I, $\rightarrow$I, $\wedge$I, $\top$I, ΠI, and $\supset$I can type only values, and are thus excluded. The context is empty, so var and fixvar are excluded, and by Property 2.5 rule contra cannot have been used.

The case for fix uses Lemma 2.14. For sub, $\vee I_{1,2}$, ΣI, $\otimes$I, $\wedge E_{1,2}$, $\supset$E and δI we simply use the IH and apply the rule again.

- **Case** $\rightarrow$E : $\mathcal{D} :: \dfrac{\cdot \vdash e_1 : A \rightarrow C \qquad \cdot \vdash e_2 : A}{\cdot \vdash e_1 e_2 : C}$

  If e_1 is not a value, use the IH with $\cdot \vdash e_1 : A \rightarrow C$, yielding e_1' such that $e_1 \mapsto e_1'$ and $\cdot \vdash e_1' : A \rightarrow C$. Given $e_1 \mapsto e_1'$ we have $e_1\, e_2 \mapsto e_1' e_2$. We have a subderivation of $\cdot \vdash e_2 : A$. By $\rightarrow$E, $\cdot \vdash e_1' e_2 : C$.

  The situation is symmetric if e_1 is a value and e_2 is not.

  If both e_1 and e_2 are values: we have $\cdot \vdash e_1 : A \rightarrow C$ and $\cdot \vdash e_2 : A$. By Lemma 2.18, $e_1 = \lambda x.\, e_0$ where, for all v' such that $\cdot \vdash v' : A$, it is the case that $\cdot \vdash [v'/x]\, e_0 : C$. Let $v' = e_2$. Then we have $\cdot \vdash [e_2/x]\, e_0 : C$. Finally, $e_1 = \lambda x.\, e_0$ and e_2 is a value, so $e_1 e_2 \mapsto [e_2/x]\, e_0$.

- **Case** $*$I : $\mathcal{D} :: \dfrac{\cdot \vdash e_1 : C_1 \qquad \cdot \vdash e_2 : C_2}{\cdot \vdash (e_1, e_2) : C_1 * C_2}$

  At least one of e_1, e_2 is not a value. Suppose e_1 is not.

	$e_1 \mapsto e_1'$ and $\cdot \vdash e_1' : C_1$	By IH
☞	$(e_1, e_2) \mapsto (e_1', e_2)$	By ev-context
	$\cdot \vdash e_2 : C_2$	Subd.
☞	$\cdot \vdash (e_1', e_2) : C_1 * C_2$	By $*$I

  The case where e_1 is a value and e_2 is not is similar.

[11] Our semantics is deterministic, so the combined form is useful. In a nondeterministic semantics, the guarantee of the combined form would be too weak: saying that *there exists* a well-typed term to which the present term steps does not preclude stepping to some other ill-typed term.

- **Case** δE : $\mathcal{D} :: \dfrac{\cdot \vdash e_0 : \delta(i) \qquad \cdot \vdash ms :_{\delta(i)} C}{\cdot \vdash \mathbf{case}\ e_0\ \mathbf{of}\ ms : C}$

  If e_0 is not a value the case is straightforward. If e_0 is a value: We have subderivations

  $$\cdot \vdash e_0 : \delta(i) \quad \text{and} \quad \cdot \vdash ms :_{\delta(i)} C$$

  By Lemma 2.18, $e_0 = c(v)$ for some v. We assume there is exactly one case arm for each constructor, so $ms = \ldots c(x) \Rightarrow e_c \ldots$ for some x, e_c. By Lemma 2.20, $x{:}A \vdash e_c : C$ where $\cdot \vdash v : A$.

  We have $\mathbf{case}\ e_0\ \mathbf{of}\ ms = \mathbf{case}\ c(v)\ \mathbf{of}\ \ldots c(x) \Rightarrow e_c \ldots$. By the reduction rule in Figure 2.4, $\mathbf{case}\ c(v)\ \mathbf{of}\ \ldots c(x) \Rightarrow e_c \ldots \mapsto_R [v/x]e_c$. By Lemma 2.14, $\cdot \vdash [v/x]\, e_c : C$. Let $e' = [v/x]\, e_c$ and we have the result.

- **Case** $*E_1$: $\mathcal{D} :: \dfrac{\cdot \vdash e_0 : C * C_2}{\cdot \vdash \mathbf{fst}(e_0) : C}$

  If e_0 is not a value, the case is straightforward. If e_0 is a value:

	$\cdot \vdash e_0 : C * C_2$	Subderivation
	$\cdot \vdash C * C_2 \leq C * C_2$	By Lemma 2.9
	$e_0 = (v_1, v_2)$	} By Lemma 2.18
☞	$\cdot \vdash v_1 : C$	} By Lemma 2.18
☞	$\mathbf{fst}((v_1, v_2)) \mapsto v_1$	} By Lemma 2.18

  The $*E_2$ case is similar.

- **Case** $\supset$E : $\mathcal{D} :: \dfrac{\cdot \vdash e : P \supset C \qquad \cdot \models P}{\cdot \vdash e : C}$

  By IH, $e \mapsto e'$ and $\cdot \vdash e' : P \supset C$. Applying $\supset$E yields $\cdot \vdash e' : C$, which was to be shown.

- **Case** $\otimes$I : $\mathcal{D} :: \dfrac{\cdot \vdash e : C' \qquad \cdot \models P}{\cdot \vdash e : P \otimes C'}$

  Similar to the previous case.

For direct, $\bot$E, $\vee$E, ΣE and $\otimes$E, which type an evaluation context $\mathcal{E}[e']$, we proceed thus:

- If the whole term $\mathcal{E}[e']$ is a value, we have the result.
- If e' is not a value:
  (1) Apply the IH to $\cdot \vdash e' : D$ to obtain $e' \mapsto e''$ with $\vdash e'' : D$.
  (2) From $e' \mapsto e''$, use ev-context to show $\mathcal{E}[e'] \mapsto \mathcal{E}[e'']$.
  (3) Reapply the rule, with premise $\vdash e'' : D$, to yield $\vdash \mathcal{E}[e''] : C$.

- If e' is a value (but $\mathcal{E}[e']$ is not), use value definiteness (Theorem 2.17), yielding a contradiction (the $\bot$E case), or a new derivation (the direct, $\vee$E, ΣE, and $\parr$E cases) which can be substituted in another premise.

The last subcase, where e' is a value and $\mathcal{E}[e']$ is not, is the most interesting; we show it for $\bot$E, $\vee$E, ΣE and $\parr$E. The direct case is similar; it uses part (v) of Theorem 2.17.

- **Case** $\bot$E : $\mathcal{D} :: \dfrac{\cdot \vdash e' : \bot \qquad \cdot \vdash \mathcal{E}[e']\ \mathsf{ok}}{\cdot \vdash \mathcal{E}[e'] : C}$

  We have $\cdot \vdash e' : \bot$ as a subderivation. By Lemma 2.9, $\cdot \vdash \bot \leq \bot$, but by Theorem 2.17 (i), $\cdot \nvdash \bot \leq \bot$, a contradiction. Therefore $\bot$E could not have derived $\mathcal{D}$.

- **Case** $\vee$E : $\mathcal{D} :: \dfrac{\begin{array}{ccc}\mathcal{D}_0 & \mathcal{D}_1 & \mathcal{D}_2 \\ \cdot \vdash e' : A \vee B & x{:}A \vdash \mathcal{E}[x] : C & y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\cdot \vdash \mathcal{E}[e'] : C}$

  e' value is given. We have $\mathcal{D}_0 :: \cdot \vdash e' : A \vee B$ as a subderivation. By Theorem 2.17 (ii), either $\cdot \vdash e' : A$ or $\cdot \vdash e' : B$ by a derivation $\mathcal{D}'$ of rank 0. Assume $\vdash e' : A$ (the other case is symmetric). $\mathcal{D}_1 :: x{:}A \vdash \mathcal{E}[x] : C$ is a subderivation of rank $\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$. By empty-$\sigma$ and pvar-σ, $\cdot \vdash e'/x : x{:}A$. By Lemma 2.16, $\mathcal{D}'_1 :: \cdot \vdash \mathcal{E}[e'] : C$ where $\mathit{Rank}(\mathcal{D}'_1) = \mathit{Rank}(\mathcal{D}_1)$, which is less than $\mathit{Rank}(\mathcal{D})$, so we can apply the IH to $\mathcal{D}'_1$, obtaining $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$.

- **Case** ΣE : $\mathcal{D} :: \dfrac{\begin{array}{cc}\mathcal{D}_0 & \mathcal{D}_1 \\ \cdot \vdash e' : \Sigma a{:}\gamma.\, A & a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C\end{array}}{\cdot \vdash \mathcal{E}[e'] : C}$

e' value	Given
$\mathcal{D}_0 :: \cdot \vdash e' : \Sigma a{:}\gamma.\, A$	Subderivation
$\exists i.\ \mathcal{D}'_0 :: \cdot \vdash e' : [i/a]A$ where $\cdot \vdash i : \gamma$ and $\mathit{Rank}(\mathcal{D}'_0) = 0$	By Theorem 2.17 (iii)
$\cdot \vdash \cdot : \cdot$	By empty-σ
$\cdot \vdash i/a : a{:}\gamma$	By ivar-σ
$\cdot \vdash i/a, e'/x : a{:}\gamma, x{:}A$	By pvar-σ
$\mathit{Rank}(i/a, e'/x) = 0$	$\mathit{Rank}(\mathcal{D}'_0) = 0$
$\mathcal{D}_1 :: a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C$	Subderivation
$\mathcal{D}'_1 :: \cdot \vdash [i/a, e'/x]\,\mathcal{E}[x] : [i/a, e'/x]C$ $\mathit{Rank}(\mathcal{D}'_1) = \mathit{Rank}(\mathcal{D}_1)$	By Lemma 2.16
$\mathcal{D}'_1 :: \cdot \vdash [e'/x]\,\mathcal{E}[x] : C$	i is a new index variable
$\mathcal{D}'_1 :: \cdot \vdash \mathcal{E}[e'] : C$	x is new, so not free in $\mathcal{E}$
$\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$	By Definition 2.15
$\mathit{Rank}(\mathcal{D}'_1) < \mathit{Rank}(\mathcal{D})$	$\mathit{Rank}(\mathcal{D}'_1) = \mathit{Rank}(\mathcal{D}_1)$
$\mu(\mathcal{D}'_1) < \mu(\mathcal{D})$	By Definition 2.15
☞ $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$	By IH

- **Case** $\parr$E : $\mathcal{D} :: \dfrac{\cdot \vdash e' : P \parr A \qquad P, x{:}A \vdash \mathcal{E}[x] : C}{\cdot \vdash \mathcal{E}[e'] : C}$

e' value	Given
$\mathcal{D}_0 :: \cdot \vdash e' : P \parr A$	Subderivation
$\mathcal{D}'_0 :: \cdot \vdash e' : A$ where $\cdot \models P$ and $\mathit{Rank}(\mathcal{D}'_0) = 0$	By Theorem 2.17 (iv)
$\cdot \vdash \cdot : \cdot$	By empty-σ
$\cdot \vdash \cdot : P$	By prop-σ
$\cdot \vdash e'/x : P, x{:}A$	By pvar-σ
$\mathit{Rank}(e'/x) = 0$	$\mathit{Rank}(\mathcal{D}'_0) = 0$
$\mathcal{D}_1 :: P, x{:}A \vdash \mathcal{E}[x] : C$	Subderivation
$\mathcal{D}'_1 :: \cdot \vdash [e'/x]\,\mathcal{E}[x] : [e'/x]C$ $\mathit{Rank}(\mathcal{D}'_1) = \mathit{Rank}(\mathcal{D}_1)$	By Lemma 2.16
$\mathcal{D}'_1 :: \cdot \vdash \mathcal{E}[e'] : C$	x is new, so not free in $\mathcal{E}$
$\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$	By Definition 2.15
$\mathit{Rank}(\mathcal{D}'_1) < \mathit{Rank}(\mathcal{D})$	$\mathit{Rank}(\mathcal{D}'_1) = \mathit{Rank}(\mathcal{D}_1)$
$\mu(\mathcal{D}'_1) < \mu(\mathcal{D})$	By Definition 2.15
☞ $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$	By IH

□

2.8 Related work

Refinement using indexed datatypes and dependent function types with indices drawn from a decidable constraint domain was proposed by Xi and Pfenning [XP99]. Their language did not introduce pure property types, requiring syntactic markers to eliminate existentials. Since this was unrealistic for many programs, Xi [Xi98] presented an algorithm allowing existential elimination at every binding site after translation to a let-normal form. Some of our results can be seen as a *post hoc* justification for that basic idea; see the remarks at the end of Section 2.4.5—and Chapter 5.

Pierce [Pie91b] gave examples of programming with intersection and union types in a pure λ-calculus using a typechecking mechanism that relied on syntactic markers. MacQueen et al. [MPS86] appear to be the first to publish work on union types, in their development of a model of recursive polymorphic types. The first systematic study of unions in a type assignment framework by

Barbanera et al. [BDCd95] identified a number of problems, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted (as it was in MacQueen et al.). That study also provided a framework for our more specialized study of a call-by-value language with possible effects.

Van Bakel et al. [vBDCdM00] showed that the minimal relevant logic B+ yields a type assignment system for the pure call-by-value λ-calculus; conjunction and disjunction become intersection and union, respectively. In their ∨-elimination rule, the subexpression may appear multiple times but must be a value; this rule is sound but impractical (see Section 2.4.1).

Litvinov [Lit03] used pure intersections and unions in typechecking programs in the object-oriented language Cecil. Igarashi and Nagira [IN06] extended Java with union types. Their type system includes both an explicit case statement (based on objects' runtime tags) and "direct access" to object members: given two classes C1 and C2, neither of which is a subclass of the other, and an `x` declared as type C1 ∨ C2, the access `x.m` is permitted as long as both C1 and C2 have a member `m`. These are not exactly tagged unions in the usual sense, nor are they exactly pure unions like those in our system—they are a form of pure union over the class hierarchy, which is *itself* a tagged union.

Finally, forms of union types have been used for control flow analysis and soft typing, as mentioned in Section 1.6.1.

2.9 Conclusion

We have designed a system of property types for the purpose of checking program invariants in call-by-value languages. We have presented the system as it was designed: incrementally, with each type constructor added orthogonally to an intermediate system that is itself sound and logically motivated. For both the definite and indefinite types, we have formulated rules that are not only sound but internally regular: the differences among ∨E, ⊥E, ΣE, direct are logical consequences of the type constructor's arity. The remarkable feature shared by all four rules is that typing may proceed in evaluation order, constituting a less *ad hoc* alternative to Xi's conversion to let-normal form. Lastly, we have formulated properties of *definiteness* of judgments, substitutions, and values, vital for our proof of type safety.

The pure type assignment system presented here is undecidable. The present system can be used to verify progress and preservation after erasure of all type annotations, and is the basis of soundness in the bidirectional system in Chapter 3. In particular, it verifies that typechecked programs do not need to carry types at runtime.

While we have elided any explicit effects from the present system for the sake of brevity, the analysis in [DP00] applies to this setting and the present system.