Combining Two Forms of Type Refinements

Jana Dunfield

September 2002[‡]
CMU-CS-02-182

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Type refinements allow invariants about algebraic datatypes to be expressed through the type system. We present a small functional language and type system that elegantly combines datasort refinements (commonly called refinement types) and dependent index refinements, so that one can specify invariants using whatever refinement is most suitable. Our type system has intersections (novel in the presence of index refinements) and restricted dependent products; we believe ML-style references and polymorphism could be added easily. As an example, we show how the type system cleanly captures several representation invariants of red-black trees.

[‡] Recompiled November 1, 2020; updates the URL in reference [18].

The author is supported in part by the National Science Foundation, under a Graduate Research Fellowship and by grant ITR/SY+SI 0121633: "Language Technology for Trustless Software Dissemination." Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Keywords: Type refinements, datasort refinements, index refinements, refinement types, dependent types

1 Introduction

Conventional static type systems such as that of Standard ML [14] enable the expression of many program invariants, allowing compilers to check those invariants at compile time. However, conventional type systems are too coarse-grained to check many desirable properties; we would like to have more *refined* type systems. Two major efforts toward this goal are the *datasort refinements* (often called *refinement types*) of Freeman, Davies, and Pfenning [10, 9, 7], and the *index refinements* of Xi and Pfenning [19, 17]. Both systems refine the *simple types* of Hindley-Milner type systems.

A datasort refinement divides the set of values inhabiting a type into several subtypes called *datasorts*, according to a description corresponding to a finite regular tree grammar. As a consequence of the grammar being regular, we can decide if a value $c(v_1, \ldots, v_n)$ belongs to some datasort by examining only c and the datasorts of its arguments $v_1, \ldots, v_n$. A value can belong to more than one datasort, and if every value of datasort δ_1 is also of datasort δ_2, we say that the first is a subsort of the second: $\delta_1 \preceq \delta_2$. A given constructor or function may have several properties: for example, if we refine lists by whether a list is of odd or even length, Cons takes even-length lists to odd-length lists and vice versa. This is done with intersection types $A \mathbin{\&} B$. We can then write the type of Cons this way:

$$\texttt{Cons} : (\mathsf{int} * \mathsf{even} \rightarrow \mathsf{odd}) \mathbin{\&} (\mathsf{int} * \mathsf{odd} \rightarrow \mathsf{even})$$

A significant limitation of datasort refinements is that they can define only a finite set of refinements—there is no way to express the property that Cons returns a list exactly one element longer than its second argument. However, Xi and Pfenning formulated a type system [19, 17] in which a restricted form of dependent typing yields a system of type refinements *without* this limitation. Types are refined by indices drawn from some constraint domain, such as integers with linear inequalities. Now the type of Cons can be written as

$$\texttt{Cons} : \Pi a{:}\mathcal{N}.\ \texttt{int} * \mathsf{list(a)} \rightarrow \mathsf{list(a+1)}$$

which may be read “for any natural number a, Cons takes an integer and a list of length a to a list of length $a + 1$.” The Π is like a universal quantifier (and can be viewed as the infinitary form of the intersection type constructor $\&$).

Xi gave several significant applications of his index refinements, such as the elimination of array-bounds checks and the verification of red-black tree invariants. The practical motivation for our work is that some properties are best expressed by datasort refinements, while others can be expressed only by index refinements. For example, the color of a node in a red-black tree is cleanly expressed by a datasort refinement, but awkwardly expressed by an index refinement: Xi had to resort to encoding the colors as integers. On the other hand, the black height of a red-black tree *cannot* be expressed by datasort refinements—doing so would require an infinite number of datasorts—but is expressed nicely by refining the type with an integer index.

We present a type system that includes both datasort refinements and index refinements, along with intersection types (novel in the presence of index refinements). The basic direction and goal of the type systems for the individual refinements we combine is not to admit more programs than a simple static type system, but fewer. Thus, each system is conservative in the sense that if a program is well typed in the simple type system and uses no type refinements, it is also well typed in the system with refinements. We share the goal of admitting fewer programs, and our approach too is conservative. Like the type systems it conceptually combines, ours is not a pure type inference system. Instead, we check bidirectionally: essentially, we either infer a type for a term or check a term against some type determined (directly or indirectly) from type annotations given by the programmer. Thus, the demands on the user are like those made by the individual refinement systems: if the user wants the compiler to guarantee additional invariants via the refined type system, she must add type annotations. Like the system of Xi and Pfenning, our type system is parametric in a constraint domain; moreover, we need only a very few properties of that domain. We prove in Section 3.7 that our system is sound.

In Section 4, we show how red-black trees can be refined in our system and examine how typechecking works for a function operating on red-black trees.

2 Related work

We have briefly described the type refinements we have combined: the datasort refinements of Freeman, Davies, and Pfenning, and the index refinements of Xi and Pfenning. Let us more closely examine that work. Datasorts were introduced by Freeman and Pfenning in 1991 [10], and Freeman gave a datasort inference algorithm (in the style of abstract interpretation) in his thesis [9]. Datasort inference is decidable and not too inefficient, but has the counterintuitive defect of inferring *too much* information. In brief, a function may have well-defined behavior on certain arguments, even though the user does not intend to ever apply the function to those arguments. Passing an unintended argument will not cause an error, so if the typechecker catches the mistake, it will not catch it where the argument was passed but at some seemingly arbitrary location. Moreover, the refined type corresponding to a function can be much longer than the type corresponding to that function's intended use (which only specifies *intended* behavior, not *all* behavior), making "type mismatch" errors more confusing. Hence the later work of Davies and Pfenning [6] focused instead on datasort *checking*, in which the user must add some type annotations explicitly giving the intended refined types. Davies has implemented a practical datasort refinement typechecker for most of Standard ML around the ML Kit compiler. In addition to an ordinary SML datatype declaration

```
datatype list = Nil | Cons of int * list
```

the user writes a refinement declaration

```
(*[ datasort odd = Cons of int * even
        and even = Nil | Cons of int * odd ]*)
```

from which the typechecker determines the types of the constructors (specifically, `Cons` has type $(\mathsf{int} * \mathsf{even} \to \mathsf{odd}) \mathbin{\&} (\mathsf{int} * \mathsf{odd} \to \mathsf{even})$ and `Nil` has type even).

Xi and Pfenning realized that restricting dependent type indices to values drawn from a decidable constraint domain yields a remarkably expressive type system, while retaining decidability of typechecking [19, 17]. Again type *inference* is avoided, so the user is obliged to add type annotations[1]. An important difficulty arises: with only the universal dependent type Π, one cannot express the types of functions in which the result's refined type is not uniquely determined by the argument's refined type. A simple example is the $\texttt{filter}(f, l)$ function on lists of integers, which returns all the elements of l such that $f(l)$ returns true. It has the simple type

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

Indexing lists by their length, the refined type should be something like

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

But we cannot fill in the blank. Xi's solution was to add a limited dependent sum type Σ that can be read as "there exists." We can then express the fact that `filter` returns a list of length m where $m \leq n$:

$$\texttt{filter} : \Pi n{:}\mathcal{N}.\, (\mathsf{int} \to \mathsf{bool}) * \mathsf{list}(n) \to (\Sigma m{:}\{\mathcal{N} \mid m \leq n\}.\, \mathsf{list}(m))$$

Xi added an index refinement checker (for the domain of integers with linear inequalities) to the Caml Light compiler. He extended the Caml datatype syntax; to obtain the type

$$\texttt{Cons} : \Pi a{:}\mathcal{N}.\, \texttt{int} * \mathsf{list}(\mathsf{a}) \to \mathsf{list}(\mathsf{a} + 1)$$

for `Cons`, one writes (notating $\Pi a{:}\mathcal{N}.$ as $\{\mathsf{a} : \texttt{nat}\}$):

```
datatype list with nat = Nil(0)
                       | {a:nat} Cons(a + 1) of int * list(a)
```

We turn now to other related work. Intersection types were introduced into the simply-typed λ-calculus by Coppo et al. [4]; their intersection type had the form (in our notation) $A_1 \mathbin{\&} \ldots \mathbin{\&} A_n \to B$, with a greatest type ω. They showed that every well-typed term has a head normal form and that a term has a normal form if and only if its type includes ω in certain positions (or not at all). Reynolds used intersection types in the explicitly-typed imperative language Forsythe [16] for several purposes, including to formulate the distinctions between readable, writable, and read/writable procedure arguments. He

[1] In Xi's examples, the type annotations are "typically" [18] less than 20% of the length of the source code.

showed that (despite the explicit typing) typechecking Forsythe is PSPACE-hard; the argument is by reduction to the evaluation of a quantified Boolean formula. The same argument applies to a language with datasort refinements, but there is no reason to suppose pathological cases arise in practice more often than in Standard ML—a language in which type inference can require superexponential time [12].

The well-known Curry-Howard isomorphism allows one to extract a program from a proof in a type theory, but the extracted program may contain logical information irrelevant to the (computational) result, making the program longer and less efficient than it should be. Hayashi [11] invented an impredicative type theory ATTT with refinement types, intersection types, and union types. His theory is designed to facilitate the exclusion of computationally irrelevant information from extracted programs.

For another purpose very different from ours, Denney [8] applied refinement types to yield a theory of structured program specification, combining conventional program-level type theory with program logic.

Cayenne [2] is a language similar to Haskell with *unrestricted* dependent type indices: any term can be used as an index. Consequently, typechecking in Cayenne is undecidable; the Cayenne typechecker "times out" if typechecking takes more than a specified number of steps. This approach appears to function reasonably well in practice. However, Cayenne does not define any kind of datatype refinement; its main concerns are with admitting *more* programs (such as a typechecked analogue of C's `printf`) rather than fewer, and with an economy of mechanism achieved by merging modules and records. It is not clear if the Cayenne approach could be extended with datatype refinements.

Systems for carrying out type inference in dynamically typed languages, known as *soft typing* [3], can reduce the runtime overhead characteristic of dynamic typing: runtime checks are inserted only where the type inference engine cannot prove the check must always succeed. Being type inference systems, these have the advantage of requiring no additional information (no type annotations) from the user. Aiken, Wimmers, and Lakshman [1] give a soft typing system with intersection, union, and conditional types. Their system infers remarkably accurate types, but fails to capture certain invariants readily expressed by datasort refinements; Davies [6] gives an example program for which it infers substantial information about a function but does *not* infer a property easily expressed through a datasort refinement.

3 The language and type system

Ours is a small functional language similar to that presented by Davies and Pfenning [7], with lambda abstraction, application, and fixed point constructs; unit and product types; **let**; refined datatypes and a simple **case** construct; and a form of explicit type annotation. The syntax is given in Figure 1. Most of these constructs require no explanation. The **case** construct cannot have nested patterns (by the grammar) and must be *non-redundant* and *exhaustive*: a **case** on a datatype τ must include exactly one arm $c \Rightarrow e$ for each constructor c of τ. Explicit type annotations allow us to annotate any term e with a type A by writing $(e : A)$.

We omit polymorphism and effects (mutable references), but we believe our type system could readily be extended to handle these, following the approach of [7] for a type system with only datasort refinements. Our system has several of that system's attributes (such as a value restriction for intersection types) precisely because those attributes avoided unsoundness in the system of datasort refinements and effects.

$$
\begin{aligned}
e ::=\ & x \mid () \\
& \mid (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e) \\
& \mid c(e) \mid \textbf{case}\ e\ \textbf{of}\ ms \\
& \mid \textbf{lam}\ x.\ e \mid e_1(e_2) \\
& \mid \textbf{fix}\ f.\ e \\
& \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \\
& \mid e : A \\
ms ::=\ & \cdot \mid c(x) \Rightarrow e \,|\, ms
\end{aligned}
$$

Figure 1: The syntax of terms.

Parameters: base index sorts s, datasorts δ, propositions P over index expressions i, j, k

$$\begin{array}{rll}
\gamma & ::= s \mid \bot \mid \mathbf{1} \mid \gamma_1 * \gamma_2 & \text{—Base, empty, unit, product sorts} \\
P & ::= \bot & \text{—Falsehood} \\
 & \mid i \doteq j & \text{—Index equality} \\
 & \mid \ldots & \\
i, j, k & ::= a & \text{—Index variables} \\
 & \mid \texttt{()} & \text{—Unit index} \\
 & \mid (i, j) \mid \mathrm{fst}(i) \mid \mathrm{snd}(i) & \text{—Pairing, left and right projection} \\
 & \mid \ldots & \\
A, B, C & ::= \mathbf{1} \mid A * B \mid A \to B & \\
 & \mid \delta(i) & \text{—Refinement by the datasort } \delta \text{ and index } i \\
 & \mid \Pi a{:}\gamma.\, A & \text{—Universal dependent types} \\
 & \mid A \,\&\, B & \text{—Intersection types}
\end{array}$$

Figure 2: Syntax of types.

The syntax of types—including index sorts γ, propositions P, indices i, j, k, and the types A, B themselves—is shown in Figure 2. The system is parametric in the base sorts s, datasorts δ, and the language of propositions P over index expressions i, j, k.

We assume that for each refined datatype we are given a relation $\preceq$ over the datasorts that is reflexive, transitive, and antisymmetric. This relation $\preceq$ is over the datasorts, not the index sorts; the actual subtyping relation, which considers index sorts as well as datasorts, is defined in Section 3.2.

As stated, the language of propositions and indices is a parameter of the system. We require only that a few constructs be present in each: the propositions P must include falsehood $\bot$ and equality of indices $i \doteq j$; the index expressions i, j, k must include index variables and pairs, along with projections fst and snd to take a pair's first and second elements. Other constructs may be included as well—we will see an example in Section 3.1 with arithmetic operations among the index expressions—but any first-order language with the aforementioned elements will fit our system.

For the types A, B, C, we have unit ($\mathbf{1}$), products $A * B$, and functions $A \to B$. We also have refinements $\delta(i)$, intersection types $A \,\&\, B$, and universal dependent types $\Pi a{:}\gamma.\, A$. (The latter are often called *dependent products*, but we prefer a name evoking universal quantification.) While $\delta(i)$ is a double refinement—by a datasort δ and by an index i—it is straightforward to utilize only one form of refinement. Suppose we have a datatype τ we wish to refine by a datasort only. For the index refinement, one simply chooses the sort **1** as the index sort; **1** is inhabited only by `()`, so the indices will always match up and it will be just as if our system had no index refinements. Likewise, if we want to refine τ by an index but do not need to refine it by a datasort, we use one datasort δ and it will be just as if we had no datasort refinements.[2]

The *simple type* of a type A is A with all refinements $\delta(i)$ replaced by the name of the type refined, all Π's erased, and all intersections $B_1 \,\&\, B_2$ replaced by the simple type of B_1 and B_2, if the simple types of B_1 and B_2 are the same. If A contains an intersection $B_1 \,\&\, B_2$ such that B_1's simple type differs from B_2's simple type, then we say that A has no simple type.

Remark 1. Davies' intersection $A \,\&\, B$ [6] is well-formed only if A and B refine the same simple type. Our intersections lack this restriction. We discuss this further in Section 3.3, in connection with the contradiction rule.

We will make use of three forms of context whose syntax is given in Figure 3: Γ which quite conventionally maps program variables x to types, φ which maps index variables a, b to index sorts and furthermore may contain propositions P representing assumptions about relationships among indices, and the constructor signature $\mathcal{S}$ which maps datatype constructors to types. No variable may appear twice in any of the contexts. Thus, we can reorder any context as we like, and we will do so without re-

[2] It seems likely that primitive types such as integers that we might wish to refine by an index, but not a datasort, could also be handled in the manner described for datatypes.

$$\begin{aligned}\Gamma &::= \cdot \mid \Gamma, x{:}A \\ \varphi &::= \cdot \mid \varphi, a{:}\gamma \mid \varphi, P \\ \mathcal{S} &::= \cdot \mid \mathcal{S}, c{:}A\end{aligned}$$

Figure 3: Syntax of program variable contexts, index variable contexts, and constructor signatures.

Property 1 (Substitution)**.** *If $\varphi \vdash i : \gamma$ and $\varphi, a{:}\gamma \models P$ then $\varphi \models [i/a]\,P$. Similarly, if $\varphi \vdash i : \gamma$ and $\varphi, a{:}\gamma \vdash j : \gamma'$ then $\varphi \vdash [i/a]\,j : \gamma'$.*

Property 2 (Weakening)**.**

(i) If $\varphi \vdash i : \gamma$ then $\varphi, a{:}\gamma' \vdash i : \gamma$, provided a is new in φ.

(ii) If $\varphi \models P$ then $\varphi, a{:}\gamma \models P$, provided a is new in φ.

Property 3. *If $\varphi \vdash i : \gamma$ and $\varphi \vdash j : \gamma$ and $\varphi \vdash k : \gamma$ for some index sort γ, then:*

(i) The relation $\varphi \models i \doteq i$ holds.

(ii) If $\varphi \models i \doteq j$ then $\varphi \models j \doteq i$.

(iii) If $\varphi \models i \doteq j$ and $\varphi \models j \doteq k$, then $\varphi \models i \doteq k$.

Property 4. *The relation $\cdot \models \bot$ does not hold.*

Property 5. $\varphi, a{:}\gamma \vdash a : \gamma$.

Property 6. *If $\varphi \models P_1$ and $\varphi, P_1 \models P_2$ then $\varphi \models P_2$.*

Figure 4: Necessary properties of the $\models$ and $\vdash$ index relations.

mark. As an example, $\cdot, a{:}\mathcal{N}, b{:}\mathcal{N}, a \doteq b + 1$ represents the assumption that a and b are natural numbers such that a equals $b + 1$. We often omit the $\cdot$.

Throughout this work, $[i/a]$ denotes a standard capture-avoiding substitution; $[i/a]\,P$ is the proposition P with all free occurrences of the variable a replaced by the index i. The application of $[i/a]$ to propositions P, indices j, types A, terms e, and variable contexts Γ is defined in the obvious way. The application of the term variable substitution $[e/x]$ to terms e' is likewise defined as one would expect.

We assume we have some procedure for deciding a satisfaction relation of the form

$$\varphi \models P$$

which can be read as "for all variables declared in φ, the propositions in φ imply P." In the examples we present, the only base sort is $\mathcal{N}$, the natural numbers. Then all inequalities contained in P must be linear, since systems of integer inequalities are undecidable in general but for linear inequalities, decision procedures such as Fourier-Motzkin [5] do exist.

It is important to note that the context φ can be inconsistent, causing *every* proposition to be satisfied. One can create an inconsistent φ by adding an index variable of the empty sort $\bot$ or by inserting an unsatisfiable proposition such as $a \doteq a + 1$. Any proposition, including falsehood ($\bot$), is then vacuously satisfied.

We also assume an appropriate index typing relation giving appropriate sorts to indices:

$$\varphi \vdash i : \gamma$$

We require that the satisfaction and index typing relations have some natural properties listed in Figure 4. Perhaps the most important of these are that we can substitute an index for a variable (Property 1), that we can weaken the context φ (Property 2), and that index equality $\doteq$ is an equivalence relation (Property 3). Each property listed is used somewhere in our proofs.

3.1 Example: bitstrings

An example of a refined datatype is strings of bits refined firstly by a datasort expressing the properties of having no leading zeroes and of being strictly positive, and secondly by an integer index denoting the bitstring's binary value. For this example, the parameters to our system are:

$$
\begin{array}{rll}
s ::= & \mathcal{N} & \text{—The natural numbers} \\
\delta ::= & \mathsf{bits} \mid \mathsf{pos} \mid \mathsf{nat} & \text{—Datasorts} \\
P ::= & \bot & \text{—Falsehood} \\
 & \mid i \doteq j & \text{—Index equality} \\
i, j, k ::= & a \mid (i, j) \mid \mathrm{fst}(i) \mid \mathrm{snd}(i) & \text{—Variables, pairing, projections} \\
 & \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots & \text{—Natural literals} \\
 & \mid i + j \mid i - j \mid i * j & \text{—Arithmetic operations}
\end{array}
$$

The datatype has three constructors ϵ, 0, and 1. We refine the datatype by datasorts bits, nat, pos:

- bits includes all bitstrings. Examples: ϵ (the empty bitstring), ϵ0, ϵ1, ϵ0010.
- nat includes all bitstrings *without leading zeros*. So ϵ is a nat, but ϵ001 and ϵ0 are not.
- pos includes every bitstring that has no leading zeros and has a nonzero binary value. Thus ϵ is not included in pos (its binary value is 0); ϵ01 is not pos (it has a leading zero); but ϵ10 is (its binary value is 2).

The subsort relation is shown in Figure 5.

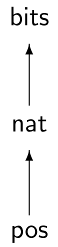

Figure 5: The subsort relation for bitstrings. It can be read "pos is a subsort of nat and nat is a subsort of bits."

The bitstrings are indexed by their binary value: ϵ10's binary value is 2, so it is indexed by 2; ϵ is indexed by 0, etc.

Figure 6 gives the signature $\mathcal{S}$ defining the types of the constructors. Recall that a type $A \,\&\, B$ means "both A and B", while a *universal dependent type* such as $\Pi a{:}\mathcal{N}.\ \ldots$ can be read as "for all indices a having sort $\mathcal{N}$ (the naturals), …".

$$
\begin{aligned}
\mathcal{S}(\epsilon) &= \mathbf{1} \to \mathsf{nat}(0) \\
\mathcal{S}(0) &= \big(\Pi a{:}\mathcal{N}.\, \mathsf{pos}(a) \to \mathsf{pos}(2{*}a)\big) \\
&\quad \& \ \big(\Pi a{:}\mathcal{N}.\, \mathsf{nat}(a) \to \mathsf{bits}(2{*}a)\big) \\
&\quad \& \ \big(\Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{bits}(2{*}a)\big) \\
\mathcal{S}(1) &= \big(\Pi a{:}\mathcal{N}.\, \mathsf{pos}(a) \to \mathsf{pos}(2{*}a{+}1)\big) \\
&\quad \& \ \big(\Pi a{:}\mathcal{N}.\, \mathsf{nat}(a) \to \mathsf{pos}(2{*}a{+}1)\big) \\
&\quad \& \ \big(\Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{bits}(2{*}a{+}1)\big)
\end{aligned}
$$

Figure 6: The signature $\mathcal{S}$ giving the types of bitstring constructors.

$$\frac{}{\varphi \vdash A \,\le\, A}\ \textbf{(refl-}\le\textbf{)} \qquad \frac{\varphi \vdash A_1 \,\le\, A_2 \qquad \varphi \vdash A_2 \,\le\, A_3}{\varphi \vdash A_1 \,\le\, A_3}\ \textbf{(trans-}\le\textbf{)}$$

$$\frac{}{\varphi \vdash A \,\&\, B \,\le\, A}\ \textbf{(sect-left-}\le\textbf{)} \qquad \frac{}{\varphi \vdash A \,\&\, B \,\le\, B}\ \textbf{(sect-right-}\le\textbf{)}$$

$$\frac{\varphi \vdash B_1 \,\le\, A_1 \qquad \varphi \vdash A_2 \,\le\, B_2}{\varphi \vdash A_1 \to A_2 \,\le\, B_1 \to B_2}\ \textbf{(arrow-}\le\textbf{)} \qquad \frac{\varphi \vdash A \,\le\, B_1 \qquad \varphi \vdash A \,\le\, B_2}{\varphi \vdash A \,\le\, B_1 \,\&\, B_2}\ \textbf{(sect-}\le\textbf{)}$$

$$\frac{\varphi \vdash A_1 \,\le\, B_1 \qquad \varphi \vdash A_2 \,\le\, B_2}{\varphi \vdash A_1 * A_2 \,\le\, B_1 * B_2}\ \textbf{(prod-}\le\textbf{)} \qquad \frac{\delta_1 \preceq \delta_2 \qquad \varphi \models i \doteq j}{\varphi \vdash \delta_1(i) \,\le\, \delta_2(j)}\ \textbf{(sort-index-}\le\textbf{)}$$

$$\frac{\varphi \vdash i : \gamma}{\varphi \vdash \Pi a{:}\gamma.\, A \,\le\, [i/a]\, A}\ \textbf{(pi-1-}\le\textbf{)} \qquad \frac{\varphi, a{:}\gamma \vdash A \,\le\, B}{\varphi \vdash A \,\le\, \Pi a{:}\gamma.\, B}\ \textbf{(pi-2-}\le\textbf{)} \ \text{ with } a \text{ not free in } A$$

Figure 7: Subtyping rules.

3.2 Subtyping

A subtyping judgment $\varphi \models A \le B$ means that A is a subtype of B under the context φ. A set of inference rules for deriving a subtyping judgment are shown in Figure 7. Subtyping should be reflexive and transitive, so we have the rules (refl-$\le$) and (trans-$\le$). If something has type $A \,\&\, B$ it should have type A and type B, so we have (sect-left-$\le$) and (sect-right-$\le$). The rules (sect-$\le$) and (prod-$\le$) are straightforward. (arrow-$\le$) is the usual subtyping rule for function types, contravariant in the argument and covariant in the result. To understand the (pi-1-$\le$) and (pi-2-$\le$) rules, recall that Π is like a universal quantifier. Thus (pi-1-$\le$) says that if i is of sort γ, we can replace a with i in A, yielding a type $[i/a]\, A$ that is a supertype of $\Pi a{:}\gamma.\, A$. For instance, $\Pi a{:}\gamma.\, \mathsf{bits}(a) \to \mathsf{bits}(2 * a)$ is the type of functions from bitstrings to bitstrings that double the value of their argument. According to (pi-1-$\le$),

$$\Pi a{:}\gamma.\, \mathsf{bits}(a) \to \mathsf{bits}(2 * a) \,\le\, \mathsf{bits}(3) \to \mathsf{bits}(2 * 3)$$

In (pi-2-$\le$), we derive that a type A is a subtype of $\Pi a{:}\gamma.\, B$ ("for all a of sort γ, B") by putting $a{:}\gamma$ into the context and deriving $A \le B$.

The (sort-index-$\le$) rule is key; it defines the subtyping relation for instances of refined datatypes. Our two kinds of refinements come together in this rule: $\delta_1(i)$ is a subtype of $\delta_2(j)$ if δ_1 is a subsort of δ_2 and i equals j.

We omit the usual distributivity rule

$$\frac{}{(A \to B) \,\&\, (A \to B') \,\le\, A \to (B \,\&\, B')}$$

which is unsound when used with side-effecting functions [7]. (The present language is pure, but we intend to incorporate effects in future work.) Moreover, without the distributivity rule, no subtyping rule contains more than one type constructor: the type constructors are orthogonal. (Davies and Pfenning [7] pointed out that this orthogonality would allow one to easily add type constructors to their system. This was borne out in the development of our system, which unlike theirs has a product type constructor.)

The rules in Figure 7 are simple but highly nondeterministic, making them hard to reason about and hard to implement. The nondeterminism of (sect-left-$\le$) and (sect-right-$\le$) is inevitable, but we can eliminate the remaining nondeterminism (for example, that arising from the (trans-$\le$) rule): following [7], we formulate a system of *algorithmic subtyping* rules (Figure 8). This system defines a $\trianglelefteq$ relation equivalent to the $\le$ relation in the sense that $A \trianglelefteq B$ holds if and only if $A \le B$ holds. A superscripted o, as in B^o, denotes that a type is "ordinary"—neither a universal type Π nor an intersection $\&$.

To see the utility of the algorithmic system, consider deriving the judgment

$$\vdash \Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{nat}(2 * a) \,\trianglelefteq\, \Pi b{:}\mathcal{N}.\, \mathsf{bits}(b) \to \mathsf{bits}(b + b)$$

In the first subtyping system, we do not know if we should apply (pi-1-$\le$), (pi-2-$\le$), or (trans-$\le$). In the algorithmic system, the only rule deriving a judgment with a Π as the supertype is (pi-2-$\trianglelefteq$). Then,

$$\frac{}{\varphi \vdash \mathbf{1} \trianglelefteq \mathbf{1}}\ \textbf{(unit-}\trianglelefteq\textbf{)}$$

$$\frac{\varphi \vdash A_1 \trianglelefteq B^o}{\varphi \vdash A_1 \,\&\, A_2 \trianglelefteq B^o}\ \textbf{(sect-left-}\trianglelefteq\textbf{)} \qquad \frac{\varphi \vdash A_2 \trianglelefteq B^o}{\varphi \vdash A_1 \,\&\, A_2 \trianglelefteq B^o}\ \textbf{(sect-right-}\trianglelefteq\textbf{)}$$

$$\frac{\varphi \vdash B_1 \trianglelefteq A_1 \qquad \varphi \vdash A_2 \trianglelefteq B_2}{\varphi \vdash A_1 \to A_2 \trianglelefteq B_1 \to B_2}\ \textbf{(arrow-}\trianglelefteq\textbf{)} \qquad \frac{\varphi \vdash A \trianglelefteq B_1 \qquad \varphi \vdash A \trianglelefteq B_2}{\varphi \vdash A \trianglelefteq B_1 \,\&\, B_2}\ \textbf{(sect-}\trianglelefteq\textbf{)}$$

$$\frac{\varphi \vdash A_1 \trianglelefteq B_1 \qquad \varphi \vdash A_2 \trianglelefteq B_2}{\varphi \vdash A_1 * A_2 \trianglelefteq B_1 * B_2}\ \textbf{(prod-}\trianglelefteq\textbf{)} \qquad \frac{\delta_1 \preceq \delta_2 \qquad \varphi \models i \doteq j}{\varphi \vdash \delta_1(i) \trianglelefteq \delta_2(j)}\ \textbf{(sort-index-}\trianglelefteq\textbf{)}$$

$$\frac{\varphi \vdash [i/a]\, A \trianglelefteq B^o \qquad \varphi \vdash i : \gamma}{\varphi \vdash \Pi a{:}\gamma.\, A \trianglelefteq B^o}\ \textbf{(pi-1-}\trianglelefteq\textbf{)} \qquad \frac{\varphi, a{:}\gamma \vdash A \trianglelefteq B}{\varphi \vdash A \trianglelefteq \Pi a{:}\gamma.\, B}\ \textbf{(pi-2-}\trianglelefteq\textbf{)} \text{ with } a \text{ not free in } A$$

Figure 8: Algorithmic subtyping rules. B^o: a type B that is not a Π or a $\&$.

when we try to derive the premise of (pi-2-$\trianglelefteq$),

$$b{:}\mathcal{N} \vdash \Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{nat}(2 * a) \trianglelefteq \mathsf{bits}(b) \to \mathsf{bits}(b + b)$$

we have a Π on the left and an ordinary type on the right, so we must use (pi-1-$\trianglelefteq$).

To prove that $\varphi \vdash A \trianglelefteq B$ if and only if $\varphi \vdash A \leq B$ (Theorem 9), we need a number of lemmas. Most of these are inversion and weakening lemmas; we also prove a substitution property and show that $\trianglelefteq$ is reflexive and transitive. With these lemmas, the proof of Theorem 9 will be straightforward.

Lemma 1 (Substitution (Subtyping))**.** *If* $\varphi, a{:}\gamma \vdash A \trianglelefteq B$ *and* $\varphi \vdash i : \gamma$ *then* $\varphi \vdash [i/a]\, A \trianglelefteq [i/a]\, B$.

Proof. By induction on the derivation $\mathcal{D}$ of $\varphi, a{:}\gamma \vdash A \trianglelefteq B$.

We show three cases; the omitted cases are similar to the last of them.

1. **Case (sort-index-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\delta_1 \preceq \delta_2 \qquad \varphi, a{:}\gamma \models j_1 \doteq j_2}{\varphi, a{:}\gamma \vdash \delta_1(j_1) \trianglelefteq \delta_2(j_2)}\ \textbf{(sort-index-}\trianglelefteq\textbf{)}}$

   $\varphi \models [i/a]\, (j_1 \doteq j_2)$ By Property 1
   $\varphi \models [i/a]\, j_1 \doteq [i/a]\, j_2$ By definition of substitution
   $\varphi \vdash \delta_1([i/a]\, j_1) \trianglelefteq \delta_2([i/a]\, j_2)$ By (sort-index-$\trianglelefteq$)
   $\varphi \vdash [i/a]\, \delta_1(j_1) \trianglelefteq [i/a]\, \delta_2(j_2)$ By definition of substitution

2. **Case (pi-1-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi, a{:}\gamma \vdash [j/b]\, A \trianglelefteq B^o \qquad \varphi, a{:}\gamma \vdash j : \gamma'}{\varphi, a{:}\gamma \vdash \Pi b{:}\gamma'.\, A \trianglelefteq B^o}\ \textbf{(pi-1-}\trianglelefteq\textbf{)}}$

   $\varphi \vdash [i/a]\, [j/b]\, A \trianglelefteq [i/a]\, B^o$ By the IH
   $\varphi \vdash [[i/a]j \,/\, b]\, [i/a]\, A \trianglelefteq [i/a]\, B^o$ By definition of substitution
   $\varphi \vdash [i/a]\, j : \gamma'$ By Property 1
   $\varphi \vdash \Pi b{:}\gamma'.\, [i/a]A \trianglelefteq [i/a]\, B^o$ By (pi-1-$\trianglelefteq$)
   $\varphi \vdash [i/a]\, \Pi b{:}\gamma'.\, A \trianglelefteq [i/a]\, B^o$ By definition of substitution

3. **Case (pi-2-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi, a{:}\gamma, b{:}\gamma' \vdash A \trianglelefteq B}{\varphi, a{:}\gamma \vdash A \trianglelefteq \Pi b{:}\gamma'.\, B}\ \textbf{(pi-2-}\trianglelefteq\textbf{)}}$

   $\varphi, b{:}\gamma' \vdash [i/a]\, A \trianglelefteq [i/a]\, B$ By the IH
   $\varphi \vdash [i/a]\, A \trianglelefteq \Pi b{:}\gamma'.\, [i/a]B$ By (pi-2-$\trianglelefteq$)
   $\varphi \vdash [i/a]\, A \trianglelefteq [i/a]\, \Pi b{:}\gamma'.\, B$ By definition of substitution

■

Lemma 2 (Inversion)**.**

(i) If $\varphi \vdash A \trianglelefteq B_1 \mathbin{\&} B_2$*, then* $\varphi \vdash A \trianglelefteq B_1$ *and* $\varphi \vdash A \trianglelefteq B_2$*.*

(ii) If $\varphi \vdash A \trianglelefteq \Pi a{:}\gamma.\, B'$*, then* $\varphi, a{:}\gamma \vdash A \trianglelefteq B'$*.*

(iii) If $\varphi \vdash A_1 \to A_2 \trianglelefteq B_1 \to B_2$*, then* $\varphi \vdash B_1 \trianglelefteq A_1$ *and* $\varphi \vdash A_2 \trianglelefteq B_2$*.*

(iv) If $\varphi \vdash A_1 * A_2 \trianglelefteq B_1 * B_2$ *then* $\varphi \vdash A_1 \trianglelefteq B_1$ *and* $\varphi \vdash A_2 \trianglelefteq B_2$*.*

(v) If $\varphi \vdash \Pi a{:}\gamma.\, A \trianglelefteq B^o$ *then there is an index* i *such that* $\varphi \vdash i : \gamma$ *and* $\varphi \vdash [i/a]\, A \trianglelefteq B^o$*.*

(vi) If $\varphi \vdash A_1 \mathbin{\&} A_2 \trianglelefteq B^o$ *is derivable then at least one of the judgments* $\varphi \vdash A_1 \trianglelefteq B^o$ *and* $\varphi \vdash A_2 \trianglelefteq B^o$ *is derivable.*

Proof. By inspection of the algorithmic subtyping rules. For example, the only rule with an intersection as the supertype is (sect-$\trianglelefteq$); the premises of that rule imply the first property. The other parts are similar. ■

Lemma 3 (φ-$\trianglelefteq$ Weakening)**.** *If* $\varphi \vdash A \trianglelefteq B$ *then for any* a *not free in* φ *and any* γ*, the judgment* $\varphi, a{:}\gamma \vdash A \trianglelefteq B$ *is derivable.*

Proof. By induction on the derivation $\mathcal{D}$ of $\varphi \vdash A \trianglelefteq B$. The proof is totally straightforward in all but four cases:

1. **Case (sort-index-$\trianglelefteq$):** Follows from Property 2(*i*) (weakening φ in the premise $\varphi \models i \doteq j$).

2. **Case (pi-1-$\trianglelefteq$):** Follows from Property 2(*ii*).

3. **Case (pi-2-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi, b{:}\gamma' \vdash A \trianglelefteq B'}{\varphi \vdash A \trianglelefteq \Pi b{:}\gamma'.\, B'}\ \textbf{(pi-2-}\trianglelefteq\textbf{)}}$

   By the IH,
   $$\varphi, b{:}\gamma', a{:}\gamma \vdash A \trianglelefteq B'$$
   is derivable. Applying (pi-2-$\trianglelefteq$) gives $\varphi, a{:}\gamma \vdash A \trianglelefteq \Pi b{:}\gamma'.\, B'$, which was to be shown.

■

Lemma 4 (& Left Introduction)**.** *If* $\varphi \vdash A \trianglelefteq B$ *then* $\varphi \vdash A \mathbin{\&} A' \trianglelefteq B$ *and* $\varphi \vdash A' \mathbin{\&} A \trianglelefteq B$*.*

Proof. By induction on the structure of B. We show the first conclusion; the proof of the second is symmetric.

1. $\boxed{B = B^o}$ We have $\varphi \vdash A \trianglelefteq B^o$. By (sect-left-$\trianglelefteq$), $\varphi \vdash A \mathbin{\&} A' \trianglelefteq B^o$.

2. $\boxed{B = \Pi b{:}\gamma.\, B'}$

$\varphi \vdash A \trianglelefteq \Pi b{:}\gamma.\, B'$	Given
$\varphi, b{:}\gamma \vdash A \trianglelefteq B'$	By Lemma 2(*ii*)
$\varphi, b{:}\gamma \vdash A \mathbin{\&} A' \trianglelefteq B'$	By the IH at B'
$\varphi \vdash A \mathbin{\&} A' \trianglelefteq \Pi b{:}\gamma.\, B'$	By (pi-2-$\trianglelefteq$)

3. $\boxed{B = B_1 \mathbin{\&} B_2}$

$\varphi \vdash A \trianglelefteq B_1 \mathbin{\&} B_2$	Given
$\varphi \vdash A \trianglelefteq B_1$	By Lemma 3
$\varphi \vdash A \mathbin{\&} A' \trianglelefteq B_1$	By the IH
$\varphi \vdash A \trianglelefteq B_2$	By Lemma 3
$\varphi \vdash A \mathbin{\&} A' \trianglelefteq B_2$	By the IH
$\varphi \vdash A \mathbin{\&} A' \trianglelefteq B_1 \mathbin{\&} B_2$	By (sect-$\trianglelefteq$)

■

Lemma 5 (Π Left Substitution)**.** *If* $\varphi \vdash [i/a]\, A \trianglelefteq B$ *and* $\varphi \vdash i : \gamma$ *then* $\varphi \vdash \Pi a{:}\gamma.\, A \trianglelefteq B$*.*

Proof. By induction on the structure of B.

1. $\boxed{B = B^o}$ Simply apply (pi-1-$\trianglelefteq$).

2. $\boxed{B = \Pi b{:}\gamma'.\, B'}$

$$\begin{array}{rl}
\varphi \vdash [i/a]\, A \ \trianglelefteq\ \Pi b{:}\gamma'.\, B' & \text{Given} \\
\varphi, b{:}\gamma' \vdash \Pi a{:}\gamma.\, A \ \trianglelefteq\ B' & \text{By Lemma 2}(ii) \\
\varphi, b{:}\gamma' \vdash [i/a]\, A \ \trianglelefteq\ B' & \text{By the IH at } B' \\
\varphi \vdash [i/a]\, \Pi a{:}\gamma.\, A \ \trianglelefteq\ \Pi b{:}\gamma'.\, B & \text{By (pi-2-}\trianglelefteq\text{)}
\end{array}$$

3. $\boxed{B = B_1 \,\&\, B_2}$

$$\begin{array}{ll}
\varphi \vdash [i/a]\, A \ \trianglelefteq\ B_1 \,\&\, B_2 & \text{Given} \\
\varphi \vdash A \ \trianglelefteq\ B_1 & \text{By Lemma 2}(i) \\
\varphi \vdash A \ \trianglelefteq\ B_2 & \text{By Lemma 2}(i) \\
\varphi \vdash [i/a]\, A \ \trianglelefteq\ B_1 & \text{By the IH at } B_1 \\
\varphi \vdash [i/a]\, A \ \trianglelefteq\ B_2 & \text{By the IH at } B_2 \\
\varphi \vdash [i/a]\, A \ \trianglelefteq\ B_1 \,\&\, B_2 & \text{By (sect-}\trianglelefteq\text{)}
\end{array}$$

■

Lemma 6 ($\trianglelefteq$ Reflexivity). *$\varphi \vdash A \ \trianglelefteq\ A$.*

Proof. By structural induction on A.

1. $\boxed{A = \delta(i)}$

$$\begin{array}{rl}
\delta \preceq \delta & \text{Reflexivity of } \preceq \\
\varphi \models i \doteq i & \text{By Property 3}(i) \\
\varphi \vdash \delta(i) \ \trianglelefteq\ \delta(i) & \text{By (sort-index-}\trianglelefteq\text{)}
\end{array}$$

2. $\boxed{A = \mathbf{1}}$

$$\begin{array}{ll}
\varphi \vdash A \ \trianglelefteq\ A & \text{By (unit-}\trianglelefteq\text{)}
\end{array}$$

3. $\boxed{A = A_1 * A_2}$

$$\begin{array}{ll}
\varphi \vdash A_1 \ \trianglelefteq\ A_1 & \text{By the IH} \\
\varphi \vdash A_2 \ \trianglelefteq\ A_2 & \text{By the IH} \\
\varphi \vdash A_1 * A_2 \ \trianglelefteq\ A_1 * A_2 & \text{By (prod-}\trianglelefteq\text{)}
\end{array}$$

4. $\boxed{A = A_1 \to A_2}$

$$\begin{array}{ll}
\varphi \vdash A_1 \ \trianglelefteq\ A_1 & \text{By the IH} \\
\varphi \vdash A_2 \ \trianglelefteq\ A_2 & \text{By the IH} \\
\varphi \vdash A_1 \to A_2 \ \trianglelefteq\ A_1 \to A_2 & \text{By (arrow-}\trianglelefteq\text{)}
\end{array}$$

5. $\boxed{A = \Pi a{:}\gamma.\, A'}$

$$\begin{array}{rl}
\varphi, a{:}\gamma \vdash A' \ \trianglelefteq\ A' & \text{By the IH} \\
\varphi, a{:}\gamma \vdash [a/a]\, A' \ \trianglelefteq\ A' & \text{By } [a/a]\, A' = A' \\
\varphi, a{:}\gamma \vdash a : \gamma & \text{By Property 5} \\
\varphi, a{:}\gamma \vdash \Pi a{:}\gamma.\, A' \ \trianglelefteq\ A' & \text{By Lemma 5} \\
\varphi \vdash \Pi a{:}\gamma.\, A' \ \trianglelefteq\ \Pi a{:}\gamma.\, A' & \text{By (pi-2-}\trianglelefteq\text{)}
\end{array}$$

6. $\boxed{A = A_1 \,\&\, A_2}$

$$\begin{array}{ll}
\varphi \vdash A_1 \ \trianglelefteq\ A_1 & \text{By the IH} \\
\varphi \vdash A_2 \ \trianglelefteq\ A_2 & \text{By the IH} \\
\varphi \vdash A_1 \,\&\, A_2 \ \trianglelefteq\ A_1 & \text{By Lemma 4} \\
\varphi \vdash A_1 \,\&\, A_2 \ \trianglelefteq\ A_2 & \text{By Lemma 4} \\
\varphi \vdash A_1 \,\&\, A_2 \ \trianglelefteq\ A_1 \,\&\, A_2 & \text{By (sect-}\trianglelefteq\text{)}
\end{array}$$

■

Lemma 7 ($*$ Right Inversion). *If $\varphi \vdash A \ \trianglelefteq\ B_1 * B_2$ then there exist A_1, A_2 such that $\varphi \vdash A \ \trianglelefteq\ A_1 * A_2$ and $\varphi \vdash A_1 \ \trianglelefteq\ B_1$ and $\varphi \vdash A_2 \ \trianglelefteq\ B_2$.*

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

Rules (sect-$\trianglelefteq$), (unit-$\trianglelefteq$), (arrow-$\trianglelefteq$), (sort-index-$\trianglelefteq$), and (pi-2-$\trianglelefteq$) cannot derive any judgment with a product on the right-hand side.

1. **Case (sect-right-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi \vdash A'_2 \ \trianglelefteq\ B_1 * B_2}{\varphi \vdash A'_1 \,\&\, A'_2 \ \trianglelefteq\ B_1 * B_2}\ \textbf{(sect-right-}\trianglelefteq\textbf{)}}$

   $\varphi \vdash A'_2 \ \trianglelefteq\ A_1 * A_2$ By the IH
   $\varphi \vdash A_1 \ \trianglelefteq\ B_1$ By the IH
   $\varphi \vdash A_2 \ \trianglelefteq\ B_2$ By the IH
   $\varphi \vdash A'_1 \,\&\, A'_2 \ \trianglelefteq\ A_1 * A_2$ By (sect-right-$\trianglelefteq$)

2. **Case (sect-left-$\trianglelefteq$):** Similar to (sect-right-$\trianglelefteq$).

3. **Case (prod-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi \vdash A_1 \ \trianglelefteq\ B_1 \qquad \varphi \vdash A_2 \ \trianglelefteq\ B_2}{\varphi \vdash A_1 * A_2 \ \trianglelefteq\ B_1 \ \trianglelefteq\ B_2}\ \textbf{(prod-}\trianglelefteq\textbf{)}}$.

   $\varphi \vdash A_1 * A_2 \ \trianglelefteq\ A_1 * A_2$ By Lemma 6

   The premises in the derivation constitute the rest of what was to be proved.

4. **Case (pi-1-$\trianglelefteq$):** $\boxed{\mathcal{D} = \dfrac{\varphi \vdash [i/a]\, A' \ \trianglelefteq\ B_1 * B_2 \qquad \varphi \vdash i : \gamma}{\varphi \vdash \Pi a{:}\gamma.\, A' \ \trianglelefteq\ B_1 * B_2}\ \textbf{(pi-1-}\trianglelefteq\textbf{)}}$

   $\varphi \vdash [i/a]\, A' \ \trianglelefteq\ A_1 * A_2$ By the IH
   $\varphi \vdash A_1 \ \trianglelefteq\ B_1$ By the IH
   $\varphi \vdash A_2 \ \trianglelefteq\ B_2$ By the IH
   $\varphi \vdash \Pi a{:}\gamma.\, A' \ \trianglelefteq\ A_1 * A_2$ By (pi-1-$\trianglelefteq$)

■

Lemma 8 ($\trianglelefteq$ Transitivity)**.** *If $\mathcal{D}_1$ derives $\varphi \vdash A_1 \ \trianglelefteq\ A_2$ and $\mathcal{D}_2$ derives $\varphi \vdash A_2 \ \trianglelefteq\ A_3$, then $\varphi \vdash A_1 \ \trianglelefteq\ A_3$.*

Proof. By structural induction on both derivations $\mathcal{D}_1, \mathcal{D}_2$.

If either derivation concluded with (unit-$\trianglelefteq$), the proof is easy: If (unit-$\trianglelefteq$) concluded $\mathcal{D}_1$ then $A_1 = \mathbf{1} = A_2$. Since $\varphi \vdash A_2 \ \trianglelefteq\ A_3$ and $A_1 = A_2$, it follows that $\varphi \vdash A_1 \ \trianglelefteq\ A_3$. Likewise, if (unit-$\trianglelefteq$) concluded $\mathcal{D}_2$ we have $A_2 = \mathbf{1} = A_3$ and so $\varphi \vdash A_1 \ \trianglelefteq\ A_2$ implies $\varphi \vdash A_2 \ \trianglelefteq\ A_3$. We have now dealt with (unit-$\trianglelefteq$).

If $\mathcal{D}_2$ ended with (sect-$\trianglelefteq$), then $A_3 = B_1 \,\&\, B_2$. From the derivation we have $\varphi \vdash A_2 \ \trianglelefteq\ B_1$ and $\varphi \vdash A_2 \ \trianglelefteq\ B_2$. By the IH, $\varphi \vdash A_1 \ \trianglelefteq\ B_1$ and $\varphi \vdash A_1 \ \trianglelefteq\ B_2$, so we apply (sect-$\trianglelefteq$) to obtain $\varphi \vdash A_1 \ \trianglelefteq\ B_1 \,\&\, B_2$.

If $\mathcal{D}_2$ ended with (pi-2-$\trianglelefteq$), then $A_3 = \Pi a{:}\gamma.\, B$. From the derivation, $\varphi, a{:}\gamma \vdash A_2 \ \trianglelefteq\ B$. a cannot be free in A. So if $\varphi \vdash A_1 \ \trianglelefteq\ A_2$, then by Lemma 3 (weakening) we have $\varphi, a{:}\gamma \vdash A_1 \ \trianglelefteq\ A_2$. Now by the IH, $\varphi, a{:}\gamma \vdash A_1 \ \trianglelefteq\ B$. The result follows by (pi-2-$\trianglelefteq$).

Now we proceed by cases on the rule concluding $\mathcal{D}_1$, omitting cases already dealt with.

1. **Case (sect-$\trianglelefteq$):** $A_1 = A$ and $A_2 = B_1 \,\&\, B_2$.

   From $\mathcal{D}_2$, $A \ \trianglelefteq\ B_1$ and $A \ \trianglelefteq\ B_2$.

   There are four rules in which the type on the left of $\trianglelefteq$ can be an intersection:

   (a) **Case (sect-left-$\trianglelefteq$):** $\boxed{\mathcal{D}_1 = \dfrac{\cdots}{\varphi \vdash A \ \trianglelefteq\ B_1 \,\&\, B_2}\ \textbf{(sect-}\trianglelefteq\textbf{)}}$ $\boxed{\mathcal{D}_2 = \dfrac{\varphi \vdash B_1 \ \trianglelefteq\ B^o}{\varphi \vdash B_1 \,\&\, B_2 \ \trianglelefteq\ B^o}\ \textbf{(sect-left-}\trianglelefteq\textbf{)}}$

   By the IH, $A \ \trianglelefteq\ B^o$.

   (b) **Case (sect-right-$\trianglelefteq$):** Similar to the preceding case.

   (c) **Case (sect-$\trianglelefteq$):** Handled above.

   (d) **Case (pi-2-$\trianglelefteq$):** Handled above.

2. **Case (sect-left-$\trianglelefteq$):** $\boxed{\mathcal{D}_1 = \dfrac{\varphi \vdash A_1' \trianglelefteq B^o}{\varphi \vdash A_1' \mathbin{\&} A_2' \trianglelefteq B^o}\ \textbf{(sect-left-}\trianglelefteq\textbf{)}}$

$\varphi \vdash B^o \trianglelefteq A_3$	Given
$\varphi \vdash A_1' \trianglelefteq A_3$	By the IH
$\varphi \vdash A_1' \mathbin{\&} A_2' \trianglelefteq A_3$	By Lemma 4

3. **Case (sect-right-$\trianglelefteq$):** Similar to (sect-left-$\trianglelefteq$).

4. **Case (arrow-$\trianglelefteq$):** $A_1 = A_1' \to A_2'$ and $A_2 = B_1 \to B_2$.
   Rules that can derive a judgment with an arrow on the left of $\trianglelefteq$:

   (a) **Case (arrow-$\trianglelefteq$):**

$$\boxed{\begin{array}{l} \mathcal{D}_1 = \dfrac{\varphi \vdash B_1 \trianglelefteq A_1' \qquad \varphi \vdash A_2' \trianglelefteq B_2}{\varphi \vdash A_1' \to A_2' \trianglelefteq B_1 \to B_2}\ \textbf{(arrow-}\trianglelefteq\textbf{)} \\ \mathcal{D}_2 = \dfrac{\varphi \vdash B_1' \trianglelefteq B_1 \qquad \varphi \vdash B_2 \trianglelefteq B_2'}{\varphi \vdash B_1 \to B_2 \trianglelefteq B_1' \to B_2'}\ \textbf{(arrow-}\trianglelefteq\textbf{)} \end{array}}$$

$\varphi \vdash B_1' \trianglelefteq A_1'$	By the IH
$\varphi \vdash A_2' \trianglelefteq B_2'$	By the IH
$\varphi \vdash A_1' \to A_2' \trianglelefteq B_1' \to B_2'$	By (arrow-$\trianglelefteq$)

   (b) **Case (sect-$\trianglelefteq$):** Handled above.

   (c) **Case (pi-2-$\trianglelefteq$):** Handled above.

5. **Case (prod-$\trianglelefteq$):** Similar to the preceding case (without the contravariance).

6. **Case (sort-index-$\trianglelefteq$):** $A_1 = \delta_1(i)$, $A_2 = \delta_2(j)$.
   Rules where the LHS can have the form $\delta_2(j)$:

   (a) **Case (sort-index-$\trianglelefteq$):** $A_3 = \delta_3(k)$.
   From $\mathcal{D}_1$ and $\mathcal{D}_2$ we have

$$\delta_1 \preceq \delta_2, \quad \varphi \models i \doteq j, \quad \delta_2 \preceq \delta_3, \quad \varphi \models j \doteq k$$

   The subsort relation is transitive so $\delta_1 \preceq \delta_3$. By transitivity of $\doteq$ (Property 3(*iii*)) we have $\varphi \models i \doteq k$. Now we simply apply (sort-index-$\trianglelefteq$).

   (b) **Case (sect-$\trianglelefteq$):** Handled above.

   (c) **Case (pi-2-$\trianglelefteq$):** Handled above.

7. **Case (pi-1-$\trianglelefteq$):** $\boxed{\mathcal{D}_1 = \dfrac{\varphi \vdash [i/a]\,A \trianglelefteq B^o \qquad \varphi \vdash i : \gamma}{\varphi \vdash \Pi a{:}\gamma.\, A \trianglelefteq B^o}\ \textbf{(pi-1-}\trianglelefteq\textbf{)}}$

$\varphi \vdash B^o \trianglelefteq A_3$	Given
$\varphi \vdash [i/a]\,A \trianglelefteq A_3$	By the IH
$\varphi \vdash \Pi a{:}\gamma.\, A \trianglelefteq A_3$	By Lemma 5

8. **Case (pi-2-$\trianglelefteq$):**

   (a) **Case (pi-1-$\trianglelefteq$):** $\boxed{\mathcal{D}_1 = \dfrac{\varphi, a{:}\gamma \vdash A_1 \trianglelefteq A'}{\varphi \vdash A_1 \trianglelefteq \Pi a{:}\gamma.\, A'}\ \textbf{(pi-2-}\trianglelefteq\textbf{)}}$ $\boxed{\mathcal{D}_2 = \dfrac{\varphi \vdash [i/a]\,A' \trianglelefteq B^o \qquad \varphi \vdash i : \gamma}{\varphi \vdash \Pi a{:}\gamma.\, A' \trianglelefteq B^o}\ \textbf{(pi-1-}\trianglelefteq\textbf{)}}$

$\varphi, a{:}\gamma \vdash A_1 \trianglelefteq A'$	From $\mathcal{D}_1$
$\varphi \vdash i : \gamma$	From $\mathcal{D}_2$
$\varphi \vdash [i/a]\,A_1 \trianglelefteq [i/a]\,A'$	By Lemma 1
$\varphi \vdash [i/a]\,A' \trianglelefteq B^o$	From $\mathcal{D}_2$
$\varphi \vdash [i/a]\,A_1 \trianglelefteq B^o$	By the IH
$[i/a]A_1 = A_1$	a is not free in A_1
$\varphi \vdash A_1 \trianglelefteq B^o$	By the previous two statements

(b) **Case (sect-$\trianglelefteq$):** Handled above.

(c) **Case (pi-2-$\trianglelefteq$):** Handled above.

■

We are now ready to state and prove Theorem 9.

Theorem 9 (Subtyping Equivalence). *$\varphi \vdash A \trianglelefteq B$ is derivable if and only if $\varphi \vdash A \leq B$ is derivable.*

Proof. We first show the rightward direction: If $\varphi \vdash A \trianglelefteq B$ then $\varphi \vdash A \leq B$. The proof is by induction on the derivation of $\varphi \vdash A \trianglelefteq B$. Cases:

1. **Case (sect-$\trianglelefteq$):** $B = B_1 \,\&\, B_2$. By the IH, $A \leq B_1$ and $A \leq B_2$. By (sect-$\leq$), $A \leq B_1 \,\&\, B_2$.
2. **Case (unit-$\trianglelefteq$):** $A = B = \mathbf{1}$. By (refl-$\leq$) $\mathbf{1} \leq \mathbf{1}$, that is, $A \leq B$.
3. **Case (sect-left-$\trianglelefteq$):** $A = A_1 \,\&\, A_2$ (and B is ordinary). By (sect-left-$\leq$), $A_1 \,\&\, A_2 \leq A_1$. By the IH, $A_1 \leq B$. Then by (trans-$\leq$), $A_1 \,\&\, A_2 \leq B$.
4. **Case (sect-right-$\trianglelefteq$):** Similar.
5. **Case (arrow-$\trianglelefteq$):** $A = A_1 \rightarrow A_2$ and $B = B_1 \rightarrow B_2$. By the IH, $B_1 \leq A_1$ and $A_2 \leq B_2$. Therefore $A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2$ by (arrow-$\leq$).
6. **Case (prod-$\trianglelefteq$):** $A = A_1 * A_2$ and $B = B_1 * B_2$. By the IH, $A_1 \leq B_1$ and $A_2 \leq B_2$. Therefore $A_1 * A_2 \leq B_1 * B_2$ by (prod-$\leq$).
7. **Case (sort-index-$\trianglelefteq$):** Immediate.
8. **Case (pi-1-$\trianglelefteq$):** $A = \Pi a{:}\gamma.\, A'$. By the IH, $[i/a]\, A' \leq B$. By (pi-1), $\Pi a{:}\gamma.\, A' \leq [i/a]\, A'$. Then by (trans-$\leq$), $\Pi a{:}\gamma.\, A' \leq B$.
9. **Case (pi-2-$\trianglelefteq$):** $B = \Pi b{:}\gamma.\, B'$. By the IH, $\varphi, b{:}\gamma \vdash A \leq B'$. By (pi-2), $\varphi \vdash A \leq \Pi b{:}\gamma.\, B'$.

With our arsenal of lemmas, the other direction is also straightforward. We must show that if $\varphi \vdash A \leq B$ then $\varphi \vdash A \trianglelefteq B$. The proof is by induction on the derivation of $\varphi \vdash A \leq B$. Cases:

1. **Case (refl-$\leq$):** $B = A$. By Lemma 6, $A \trianglelefteq A$.
2. **Case (trans-$\leq$):** $A = A_1$ and $B = A_3$. By the IH, $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_3$. By Lemma 8, $A_1 \trianglelefteq A_3$.
3. **Case (sect-left-$\leq$):** Follows from Lemma 4.
4. **Case (sect-right-$\leq$):** Follows from Lemma 4.
5. **Case (sect-$\leq$):** By the IH, $A \trianglelefteq B_1$ and $A \trianglelefteq B_2$. By (sect-$\trianglelefteq$), $A \trianglelefteq B_1 \,\&\, B_2$.
6. **Case (arrow-$\leq$):** Symmetric to the arrow-Sub case for the other direction.
7. **Case (prod-$\leq$):** Symmetric to the prod-Sub case for the other direction.
8. **Case (pi-1-$\leq$):** $A = \Pi a{:}\gamma.\, A'$. By Lemma 6, $[i/a]\, A \trianglelefteq [i/a]\, A$. We have $\varphi \vdash i : \gamma$ from (pi-1). Then by Lemma 5, $\Pi a{:}\gamma.\, A' \trianglelefteq B$.
9. **Case (pi-2-$\leq$):** $B = \Pi b{:}\gamma.\, B'$. By the IH, $\varphi, a{:}\gamma \vdash A \trianglelefteq B'$. By (pi-2-$\trianglelefteq$), $\varphi \vdash A \trianglelefteq \Pi b{:}\gamma.\, B'$.
10. **Case (sort-index-$\leq$):** Symmetric to the case in the other direction.

■

3.3 Typing

We make no attempt to infer all types; instead, our type system is *bidirectional*, with two judgment forms. In the first, $\varphi; \Gamma \vdash e \uparrow A$, we *infer* for e a type A. In the second, $\varphi; \Gamma \vdash e \downarrow A$, we *check* e against the type A. In many cases, the programmer must explicitly write type annotations $e : A$. Before giving the typing rules, we must define the values v, as some of our typing rules are restricted to giving types to values (for reasons discussed in Section 3.3.1).

$$\begin{aligned} v ::= {} & x \mid () \mid (v_1, v_2) \\ & \mid \textbf{lam}\, x.\, e \\ & \mid c(v) \end{aligned}$$

The typing rules are given in Figure 9. The rules (cons), (matches), (match-δ-c) and (case) are a function of the signature $\mathcal{S}$ giving types to constructors of refined datatypes. (match-δ-c) represents a family of inference rules, one for each datasort/constructor pair. The judgments it derives, which have the form

$$\varphi; \Gamma \vdash c(x) \Rightarrow e \downarrow_{\delta(i)} C$$

should be read as "e typechecks against C, assuming $c(x)$ has type subsort $\delta(i)$." Likewise, the judgment form

$$\varphi; \Gamma \vdash ms \downarrow_{\delta(i)} C$$

should be read as "all the expressions in ms typecheck against C, assuming the value being matched has type $\delta(i)$."

Remark 2. The (sect-up-1) rule seems to do exactly the same thing as (sect-left-$\trianglelefteq$) when used to derive (sub-down)'s premise, but it is not redundant. The distinction is that (sect-up-1) derives an inference judgment ($\varphi; \Gamma \vdash e \uparrow A$) whereas (sub-down) derives a checking judgment ($\varphi; \Gamma \vdash e \downarrow A$). We have (sect-up-2) and (pi-elim) for the same reason.

We forbid the application of (case) unless the case expression includes exactly one arm $c_i \Rightarrow e_i$ for every constructor of the type τ refined by δ. Moreover, to simplify the proofs involving these rules, we require that each of the types in the signature $\mathcal{S}$ be an intersection $A_1 \,\&\, \dots \,\&\, A_m$ where each conjunct A_k has the form

$$\Pi a_k{:}\gamma_k.\, L_k \to \delta_k(j_k)$$

This restriction enables us to have just one rule, (cons), for typing an unapplied constructor c. It also yields a relatively straightforward formulation of the (match-δ-c) rules. We define the various symbols appearing in (matches) as follows: Let $A_{\delta 1}, \dots, A_{\delta n}$ be those conjuncts in $\mathcal{S}(c)$ such that δ_k (the sort to the right of the arrow) is a subsort of δ, and let $a_{\delta k}$, $\gamma_{\delta k}$, $L_{\delta k}$, $\delta_{\delta k}$, $j_{\delta k}$ be defined by

$$A_{\delta k} = \Pi a_{\delta k}{:}\gamma_{\delta k}.\, L_{\delta k} \to \delta_{\delta k}(j_{\delta k})$$

Let $L_{\delta k}$ be the type to the left of the arrow in $A_{\delta k}$, for $1 \leq k \leq n$. Likewise, let $\delta_k(j_k)$ be the type to the right of the arrow in $A_{\delta k}$.

An example will clarify how we construct the (match-δ-c) rule family. Consider the bitstring constructor $\mathsf{0}$. From Figure 6, its type $\mathcal{S}(\mathsf{0})$ is given by

$$\begin{aligned} \mathcal{S}(\mathsf{0}) = {} & \big(\Pi a{:}\mathcal{N}.\, \mathsf{pos}(a) \to \mathsf{pos}(2{*}a)\big) \\ \& \ & \big(\Pi a{:}\mathcal{N}.\, \mathsf{nat}(a) \to \mathsf{bits}(2{*}a)\big) \\ \& \ & \big(\Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{bits}(2{*}a)\big) \end{aligned}$$

Thus, according to the above definitions, we have

$$\begin{aligned} A_{\mathsf{bits}1} &= \Pi a{:}\mathcal{N}.\, \mathsf{pos}(a) \to \mathsf{pos}(2{*}a) \\ A_{\mathsf{bits}2} &= \Pi a{:}\mathcal{N}.\, \mathsf{nat}(a) \to \mathsf{bits}(2{*}a) \\ A_{\mathsf{bits}3} &= \Pi a{:}\mathcal{N}.\, \mathsf{bits}(a) \to \mathsf{bits}(2{*}a) \end{aligned}$$

Since every datasort is a subsort of the "top" datasort bits (Figure 5) A_{bits} has all the conjuncts of $\mathcal{S}(\mathsf{0})$. More interestingly, we also have

$$A_{\mathsf{nat}1} = \Pi a{:}\mathcal{N}.\, \mathsf{pos}(a) \to \mathsf{pos}(2{*}a)$$

$$\frac{\varphi;\Gamma\vdash e\uparrow A\,\&\,B}{\varphi;\Gamma\vdash e\uparrow A}\ \textbf{(sect-up-1)}\qquad\frac{\varphi;\Gamma\vdash e\uparrow A\,\&\,B}{\varphi;\Gamma\vdash e\uparrow B}\ \textbf{(sect-up-2)}\qquad\frac{\varphi;\Gamma\vdash v\downarrow A_1\qquad\varphi;\Gamma\vdash v\downarrow A_2}{\varphi;\Gamma\vdash v\downarrow A_1\,\&\,A_2}\ \textbf{(sect-down)}$$

$$\frac{\varphi;\Gamma\vdash e\uparrow \Pi a{:}\gamma.\,A\qquad\varphi\vdash i:\gamma}{\varphi;\Gamma\vdash e\uparrow [i/a]\,A}\ \textbf{(pi-elim)}\qquad\frac{\varphi,a{:}\gamma;\Gamma\vdash v\downarrow A}{\varphi;\Gamma\vdash v\downarrow \Pi a{:}\gamma.\,A}\ \textbf{(pi-intro)}\qquad\frac{\varphi\models\bot}{\varphi;\Gamma\vdash e\downarrow A}\ \textbf{(contra-down)}$$

$$\frac{\Gamma(x)=A}{\varphi;\Gamma\vdash x\uparrow A}\ \textbf{(var-up)}\qquad\frac{\varphi;\Gamma\vdash e\uparrow A\qquad\varphi\vdash A\ \trianglelefteq\ B}{\varphi;\Gamma\vdash e\downarrow B}\ \textbf{(sub-down)}$$

$$\frac{\varphi;\Gamma\vdash e_1\uparrow A\to B\qquad\varphi;\Gamma\vdash e_2\downarrow A}{\varphi;\Gamma\vdash e_1(e_2)\uparrow B}\ \textbf{(app-up)}\qquad\frac{\varphi;\Gamma\vdash e_1\uparrow B\qquad\varphi;\Gamma,x{:}B\vdash e_2\downarrow A}{\varphi;\Gamma\vdash \textbf{let }x=e_1\textbf{ in }e_2\textbf{ end}\downarrow A}\ \textbf{(let-down)}$$

$$\frac{\varphi;\Gamma\vdash e\downarrow A}{\varphi;\Gamma\vdash (e:A)\uparrow A}\ \textbf{(anno-up)}\qquad\frac{\varphi;\Gamma,f{:}A\vdash e\downarrow A}{\varphi;\Gamma\vdash \textbf{fix }f.\,e\downarrow A}\ \textbf{(fix-down)}$$

$$\frac{\varphi;\Gamma\vdash e\uparrow A*B}{\varphi;\Gamma\vdash \pi_1(e)\uparrow A}\ \textbf{(fst-up)}\qquad\frac{\varphi;\Gamma\vdash e\uparrow A*B}{\varphi;\Gamma\vdash \pi_2(e)\uparrow B}\ \textbf{(snd-up)}\qquad\frac{}{\varphi;\Gamma\vdash ()\downarrow \mathbf{1}}\ \textbf{(unit-down)}$$

$$\frac{\varphi;\Gamma\vdash e_1\downarrow A_1\qquad\varphi;\Gamma\vdash e_2\downarrow A_2}{\varphi;\Gamma\vdash (e_1,e_2)\downarrow A_1*A_2}\ \textbf{(prod-down)}\qquad\frac{\varphi;\Gamma,x{:}A\vdash e\downarrow B}{\varphi;\Gamma\vdash \textbf{lam }x.\,e\downarrow A\to B}\ \textbf{(lam)}$$

$$\frac{\mathcal{S}(c)=\big(\Pi a_1{:}\gamma_1.\,L_1\to\delta_1(j_1)\big)\,\&\,\cdots\,\&\,\big(\Pi a_n{:}\gamma_n.\,L_n\to\delta_n(j_n)\big)\qquad\varphi\vdash i:\gamma_\ell}{\varphi\vdash c:[i/a_\ell]\,L_\ell\to\delta_\ell(j_\ell)}\ \textbf{(cons)}\ 1\le\ell\le n$$

$$\frac{\varphi\vdash c:A\to\delta_2(i)\qquad\varphi\vdash\delta_2(i)\ \trianglelefteq\ \delta_1(j)\qquad\varphi;\Gamma\vdash e\downarrow A}{\varphi;\Gamma\vdash c(e)\downarrow\delta_1(j)}\ \textbf{(cons-app)}$$

$$\frac{\big(\varphi,a_{\delta k}{:}\gamma_{\delta k},i\doteq j_{\delta k};\Gamma,x{:}L_{\delta k}\vdash e\downarrow C\big)_k}{\varphi;\Gamma\vdash c(x)\Rightarrow e\ \downarrow_{\delta(i)}C}\ \textbf{(match-}\delta\text{-}c\textbf{)}\ \text{first premise is repeated for appropriate } k \text{ (see text)}$$

$$\frac{\varphi;\Gamma\vdash c(x)\Rightarrow e\ \downarrow_{\delta(i)}C\qquad ms\downarrow_{\delta(i)}C}{\varphi;\Gamma\vdash c(x)\Rightarrow e\mid ms\ \downarrow_{\delta(i)}C}\ \textbf{(matches)}\qquad\frac{}{\varphi;\Gamma\vdash\cdot\downarrow_{\delta(i)}C}\ \textbf{(null-matches)}$$

$$\frac{\varphi;\Gamma\vdash e\uparrow\delta(i)\qquad ms\downarrow_{\delta(i)}C}{\varphi;\Gamma\vdash\textbf{case }e\textbf{ of }ms\ \downarrow\ C}\ \textbf{(case)}$$

Figure 9: Typing rules. See the text for an explanation of (cons), (match-δ-c), (matches), and (case).

This is the only A_{nat} type, for the datasort bits appearing on the right hand side of the arrow in the other two conjuncts of $\mathcal{S}(0)$ is not a subsort of nat. Finally, A_{pos}'s is

$$A_{\mathsf{pos1}}=\Pi a{:}\mathcal{N}.\,\mathsf{pos}(a)\to\mathsf{pos}(2*a)$$

From A_{pos} we generate the premises of (match-pos-0). We have

$$L_{\mathsf{pos1}}=\mathsf{pos}(a),\quad a_{\mathsf{pos1}}=a,\quad \gamma_{\mathsf{pos1}}=\mathcal{N},\quad j_{\mathsf{pos1}}=2*a,\quad \delta_{\mathsf{pos1}}=\mathsf{pos}(a)$$

Thus, the resulting rule is

$$\frac{\varphi,a{:}\mathcal{N},i\doteq 2*a;\Gamma,x{:}\mathsf{pos}(a)\vdash e\downarrow C}{\varphi;\Gamma\vdash e0\downarrow_{\mathsf{pos}}C}\ \textbf{(match-}\mathsf{pos}\textbf{-0)}$$

The entire family of (match-δ-c) rules for bitstrings is given in Figure 11.

We can now justify the presence of (contra-down) through an example. It is quite sensible that, if b has type bits(0), the expression

$$\textbf{case }b\textbf{ of }\epsilon\Rightarrow\epsilon 1\ \mid x0\Rightarrow\epsilon 1\ \mid y1\Rightarrow\epsilon 01$$

should check against the type $\mathsf{nat}(1)$: The last case arm constructs a value $\epsilon 01$ with a leading zero, but that arm is unreachable since b is indexed by 0 and any value of the form $y1$ is not indexed by 0. By the construction of the (match-δ-c) rules, we do not check arms that are unreachable by virtue of an impossible *datasort* refinement: in (match-δ-c) we generate no premises corresponding to c if none of the datasorts on the right hand side of the arrows in $\mathcal{S}(c)$ are subsorts of δ. The contradiction rule effectively excludes from typechecking arms made unreachable by virtue of an impossible *index* refinement: If the equation $i \doteq j_{\delta k}$ does not hold under φ then $\varphi, \ldots, i \doteq j_{\delta k} \models \bot$. Therefore we can apply (contra-down) to derive the premise

$$\varphi, a_{\delta k}{:}\gamma_{\delta k}, i \doteq j_{\delta k}; \Gamma, x{:}L_{\delta k} \vdash e' \downarrow C$$

For the example above, the relevant premise in (match-bits-pos) is

$$\varphi, a{:}\mathcal{N}, 0 \doteq 2{*}a{+}1; \Gamma, x{:}L_{\delta k} \vdash \epsilon 01 \downarrow \mathsf{nat}(1)$$

There is no natural number a such that $0 = 2a + 1$, so the relation $\varphi, a{:}\mathcal{N}, 0 \doteq 2{*}a{+}1 \models \bot$ holds and we can apply (contra-down).

One can take unreachability too far: we probably do not want

$$\textbf{case}\ \epsilon\ \textbf{of}\ \epsilon \Rightarrow \epsilon 1\ \mid x0 \Rightarrow ()\ \mid y1 \Rightarrow ()$$

to be well-typed, despite the fact that the second and third arms will never be evaluated. We see two justifications for banning such expressions. The first is that allowing them to typecheck seems simply nonsensical. The second is that, for our refinements to be a *conservative extension* of some language (say SML), every program that is well-typed in the language with refinements added should be well-typed in unrefined SML. This is consistent with our goal: to catch more bugs, not to admit more programs[3]; moreover, programmers can then correctly understand the type system as an already-familiar system which has been augmented with refinements. Therefore, a real compiler should use a two-phase strategy: First typecheck the program in an unrefined type system, with both kinds of refinements erased; if simple typechecking succeeds, invoke the refined typechecker. In this way, the programs admitted are those in the shaded region in Figure 10.

Note that this two-phase strategy excludes programs making use of intersection types in a way not allowed by a refinement restriction (which restricts the types A, B in $A \mathbin{\&} B$ to be refinements of the same simple type). Thus, the practical strategy necessitated by the contradiction rule leads to the exclusion of these programs, which inhabit the regions marked & in Figure 10, despite the fact that our formal system has no refinement restriction.

3.3.1 The value restriction

In our formulation, the typing rules (sect-down) and (pi-intro), which are respectively the introduction rules for & and Π, can only type values v. Without this restriction, two problems would appear—one in connection with both type constructors, one with Π.

First, we could not extend the language with effects, for Davies and Pfenning demonstrated [7] that combining mutable references and intersection types leads to unsoundness unless the intersection introduction rule is restricted to values. (The issue has similarities to that of polymorphism, which led to a value restriction being added to Standard ML [14].) Briefly, one can put a value of a subsort δ in a reference cell, write a value of a supersort δ' of δ to the cell, and then read from the cell under the assumption that the cell contains something of sort δ.

Second, if (pi-intro) is not restricted to values, some non-values are well-typed but irreducible. An example:

$$\textbf{let}\ x = (()\,()\ : \Pi a{:}\bot.\, A)\ \textbf{in}\ e\ \textbf{end}$$

Suppose (pi-intro) could type non-values. In the premise of (pi-intro) we have

$$a{:}\bot; \cdot \vdash ()\,() \downarrow A$$

$a{:}\bot$ is inconsistent (that is, $a{:}\bot \models \bot$), so the premise is immediately derivable by applying (contra-down). But $()\,()$ is not a value and cannot be reduced. Thus, we lose type safety if (pi-intro) is not restricted to typing values.

[3] In contrast, the main goal of the Cayenne type system is to (safely) admit more programs. Augustsson [2] shows some situations in which admitting more programs is reasonable and desirable, but unreachable-case-arm programs like the above seem useless.

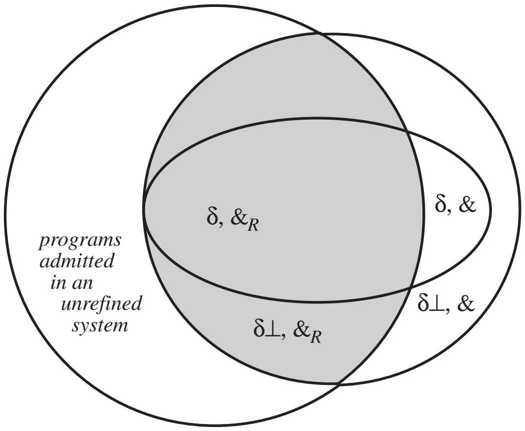

Figure 10: Set relationship of admitted programs, with and without refinements (δ), refinements with the (contra-down) rule ($\delta\bot$), unrestricted intersection (&), and intersection restricted to types refining the same simple type ($\&_R$). The leftmost circle contains those programs admitted in a simple type system with no refinements and no form of intersection. The shaded area represents those programs admitted if we first do simple typechecking (rejecting all programs outside the leftmost circle).

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq 0; \Gamma \vdash e \downarrow C}{\varphi; \Gamma \vdash \epsilon \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{bits}\textbf{-}\epsilon) \qquad \frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a; \Gamma, x{:}\mathsf{bits}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash x\mathsf{0} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{bits}\textbf{-}\mathsf{0})$$

$$\frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a{+}1; \Gamma, y{:}\mathsf{bits}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash y\mathsf{1} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{bits}\textbf{-}\mathsf{1})$$

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq 0; \Gamma \vdash e \downarrow C}{\varphi; \Gamma \vdash \epsilon \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{nat}\textbf{-}\epsilon) \qquad \frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a; \Gamma, x{:}\mathsf{pos}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash x\mathsf{0} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{nat}\textbf{-}\mathsf{0})$$

$$\frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a{+}1; \Gamma, y{:}\mathsf{nat}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash y\mathsf{1} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{nat}\textbf{-}\mathsf{1})$$

$$\frac{}{\varphi; \Gamma \vdash \epsilon \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{pos}\textbf{-}\epsilon) \qquad \frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a; \Gamma, x{:}\mathsf{pos}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash x\mathsf{0} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{pos}\textbf{-}\mathsf{0})$$

$$\frac{\varphi, a{:}\mathcal{N}, i \doteq 2{*}a{+}1; \Gamma, y{:}\mathsf{nat}(a) \vdash e \downarrow C}{\varphi; \Gamma \vdash y\mathsf{1} \Rightarrow e \downarrow_{\mathsf{bits}} C}\ (\textbf{match-}\mathsf{pos}\textbf{-}\mathsf{1})$$

Figure 11: The family of (match-δ-c) rules for bitstrings.

$$\frac{\varphi;\Gamma \vdash v : A_1 \qquad \varphi;\Gamma \vdash v : A_2}{\varphi;\Gamma \vdash v : A_1 \mathbin{\&} A_2}\ \textbf{(sect-down)} \qquad \frac{\varphi, a{:}\gamma; \Gamma \vdash v : A}{\varphi;\Gamma \vdash v : \Pi a{:}\gamma.\, A}\ \textbf{(pi-intro)}$$

$$\frac{\Gamma(x) = A}{\varphi;\Gamma \vdash x : A}\ \textbf{(var)} \qquad \frac{\varphi;\Gamma \vdash e : A \qquad \varphi \vdash A \trianglelefteq B}{\varphi;\Gamma \vdash e : B}\ \textbf{(sub)} \qquad \frac{\varphi \models \bot}{\varphi;\Gamma \vdash e : A}\ \textbf{(contra)}$$

$$\frac{\varphi;\Gamma \vdash e_1 : A \to B \qquad \varphi;\Gamma \vdash e_2 : A}{\varphi;\Gamma \vdash e_1(e_2) : B}\ \textbf{(app)} \qquad \frac{\varphi;\Gamma \vdash e_1 : B \qquad \varphi;\Gamma, x{:}B \vdash e_2 : A}{\varphi;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} : A}\ \textbf{(let)}$$

$$\frac{\varphi;\Gamma, f{:}A \vdash e : A}{\varphi;\Gamma \vdash \textbf{fix}\ f.\, e : A}\ \textbf{(fix)}$$

$$\frac{\varphi;\Gamma \vdash e : A * B}{\varphi;\Gamma \vdash \pi_1(e) : A}\ \textbf{(fst)} \qquad \frac{\varphi;\Gamma \vdash e : A * B}{\varphi;\Gamma \vdash \pi_2(e) : B}\ \textbf{(snd)} \qquad \frac{}{\varphi;\Gamma \vdash () : \mathbf{1}}\ \textbf{(unit)}$$

$$\frac{\varphi;\Gamma \vdash e_1 : A_1 \qquad \varphi;\Gamma \vdash e_2 : A_2}{\varphi;\Gamma \vdash (e_1, e_2) : A_1 * A_2}\ \textbf{(prod)} \qquad \frac{\varphi;\Gamma, x{:}A \vdash e : B}{\varphi;\Gamma \vdash \textbf{lam}\ x.\, e : A \to B}\ \textbf{(lam)}$$

$$\frac{\varphi \vdash c : A \to \delta_2(i) \qquad \varphi \vdash \delta_2(i) \trianglelefteq \delta_1(j) \qquad \varphi;\Gamma \vdash e : A}{\varphi;\Gamma \vdash c(e) : \delta_1(j)}\ \textbf{(cons-app)}$$

$$\frac{\mathcal{S}(c) = \big(\Pi a_1{:}\gamma_1.\, L_1 \to \delta_1(j_1)\big) \mathbin{\&} \cdots \mathbin{\&} \big(\Pi a_n{:}\gamma_n.\, L_n \to \delta_n(j_n)\big) \qquad \varphi \vdash i : \gamma_\ell}{\varphi \vdash c : [i/a_\ell]\, L_\ell \to \delta_\ell(j_\ell)}\ \textbf{(cons)}\ 1 \leq \ell \leq n$$

$$\frac{\big(\varphi, a_{\delta k}{:}\gamma_{\delta k}, i \doteq j_{\delta k}; \Gamma, x{:}L_{\delta k} \vdash e : C\big)_k}{\varphi;\Gamma \vdash c(x) \Rightarrow e \downarrow_{\delta(i)} C}\ \textbf{(match-}\boldsymbol{\delta}\textbf{-}\boldsymbol{c}\textbf{)}\ \text{first premise is repeated for appropriate } k$$

$$\frac{\varphi;\Gamma \vdash c(x) \Rightarrow e \downarrow_{\delta(i)} C \qquad ms \downarrow_{\delta(i)} C}{\varphi;\Gamma \vdash c(x) \Rightarrow e \mid ms \downarrow_{\delta(i)} C}\ \textbf{(matches)} \qquad \frac{}{\varphi;\Gamma \vdash \cdot \downarrow_{\delta(i)} C}\ \textbf{(null-matches)}$$

$$\frac{\varphi;\Gamma \vdash e : \delta(i) \qquad ms \downarrow_{\delta(i)} C}{\varphi;\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms : C}\ \textbf{(case)}$$

Figure 12: Rules for the (undirected) type assignment system. See Section 3.3 for an explanation of (cons), (match-δ-c), (matches), and (case).

3.4 A type assignment system

To simplify the proof of type safety, we introduce an undirected type system (Figure 12). This system is almost the same as the bidirectional system, with all $\downarrow$- and $\uparrow$-judgments changed to :-judgments. However, it lacks the rules (sect-up-1), (sect-up-2), and (pi-elim), which are not needed: the rule (sub-down) is now an undirected rule (sub), so that the omitted rules can be derived from (sub) and the subtyping rules (using Lemmas 4 and 5). Here, we are not interested in the practicality of type-checking, so we can omit the rule (anno-up) (which is necessary since the typechecker cannot "guess" type annotations).

The type assignment system is related to the one given in Figure 9 in the following sense:

Theorem 10 (Portability of Typing)**.** *Let* $||e||$ *denote the term* e *with type annotations erased, that is, with all occurrences of* $(e : A)$ *replaced by* e*.*

If $\mathcal{D}$ *is a derivation of* $\varphi;\Gamma \vdash e \downarrow A$*, then there exists a derivation* $\mathcal{D}'$ *of* $\varphi;\Gamma \vdash ||e|| : A$*.*

Proof. To obtain $\mathcal{D}'$ from $\mathcal{D}$:

(1) erase the type annotations and replace every $\uparrow$ and $\downarrow$ by :,

(2) remove uses of (anno-up), which now have the form

$$\frac{\begin{array}{c}\vdots\\ \varphi;\Gamma\vdash e:A\end{array}}{\varphi;\Gamma\vdash e:A}$$

(3) convert uses of (sect-up-1), (sect-up-2), and (pi-elim) into uses of (sub), as

$$\frac{\begin{array}{c}\vdots\\ \varphi;\Gamma\vdash e:A\,\&\,B\end{array}}{\varphi;\Gamma\vdash e:A} \qquad\longrightarrow\qquad \frac{\begin{array}{c}\vdots\\ \varphi;\Gamma\vdash e:A\,\&\,B\end{array} \qquad \varphi\vdash A\,\&\,B\ \trianglelefteq\ A}{\varphi;\Gamma\vdash e:A}$$

The derivability of the subtyping judgment follows from Theorem 9 and the rules (sect-left-$\leq$), (sect-right-$\leq$), and (pi-1-$\leq$).

■

Because any $\uparrow$-judgment can be turned into a $\downarrow$-judgment by applying (sub-down) with second premise $\varphi\vdash A\ \trianglelefteq\ A$, we also have

Corollary 11. *If $\varphi;\Gamma\vdash e\uparrow A$ is derivable, then $\varphi;\Gamma\vdash ||e|| : A$ is derivable.*

We will prove that this system, not the bidirectional system, is sound with respect to the dynamic semantics that follows. The bidirectionality was to make typechecking practical, and now only complicates matters (particularly the proof of safety). In addition, we want to allow types to be erased by runtime, and therefore want a dynamic semantics in which types do not appear. Programs typed by the type assignment system have no type annotations, so the type assignment system fits such a semantics.

3.5 Dynamic semantics

A deterministic call-by-value dynamic semantics over *type-erased* terms is defined by the rules in Figure 13. Since the terms have all their type annotations erased, there is no rule for $(e : A)$. Note that the case expression in (ev-case) must have exactly one arm for each constructor.

Theorem 12 (Determinism)**.** *If $e\mapsto e'$ and $e\mapsto e''$, it must be the case that $e' = e''$.*

Proof. Assume $e\mapsto e'$. The proof is by cases on the form of e; we give two representative cases.

1. $\boxed{e = \pi_1(e'')}$ If e'' is a value and $\pi_1(e'')$ steps to e', it must do so by (ev-fst). Otherwise, e'' is not a value and so $\pi_1(e'')$ can step only by (ev-fst-arg). In either case, at most one rule can be applied, so e' is uniquely determined.

2. $\boxed{e = e_1(e_2)}$ If e_1 is not a value and $e_1(e_2)\mapsto e'$, it must do so by (ev-app-fn). If e_1 is a value but e_2 is not, $e_1(e_2)\mapsto e'$ only by (ev-app-arg). (Note that an unapplied constructor is not an expression, so we do not consider $e = c(e_2)$ here.) If both e_1 and e_2 are values, the only rule that could possibly apply is (ev-beta). In all cases, e' is uniquely determined.

■

3.6 Lemmas for type safety

Lemma 13 (Substitution of an index variable)**.** *If $\varphi, a{:}\gamma;\Gamma\vdash e : A$ and $\varphi\vdash i:\gamma$ then $\varphi;[i/a]\Gamma\vdash e : [i/a]\,A$.*

Proof. By induction on the derivation $\mathcal{D}$ of $\varphi, a{:}\gamma;\Gamma\vdash e : A$. The only interesting case is for the rule (contra).

$$\frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)}\ \textbf{(ev-pair-1)} \qquad \frac{e_2 \mapsto e_2'}{(v, e_2) \mapsto (v, e_2')}\ \textbf{(ev-pair-2)}$$

$$\frac{}{\pi_1(v_1, v_2) \mapsto v_1}\ \textbf{(ev-fst)} \qquad \frac{}{\pi_2(v_1, v_2) \mapsto v_2}\ \textbf{(ev-snd)}$$

$$\frac{e \mapsto e'}{\pi_1(e) \mapsto \pi_1(e')}\ \textbf{(ev-fst-arg)} \qquad \frac{e \mapsto e'}{\pi_2(e) \mapsto \pi_2(e')}\ \textbf{(ev-snd-arg)}$$

$$\frac{e_1 \mapsto e_1'}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} \mapsto \textbf{let}\ x = e_1'\ \textbf{in}\ e_2\ \textbf{end}}\ \textbf{(ev-let-arg)} \qquad \frac{}{\textbf{let}\ x = v\ \textbf{in}\ e_2\ \textbf{end} \mapsto [v/x]\, e_2}\ \textbf{(ev-let)}$$

$$\frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}\ \textbf{(ev-app-fn)} \qquad \frac{e_2 \mapsto e_2'}{v(e_2) \mapsto v(e_2')}\ \textbf{(ev-app-arg)} \qquad \frac{e \mapsto e'}{c(e) \mapsto c(e')}\ \textbf{(ev-cons-arg)}$$

$$\frac{}{(\textbf{lam}\ x.\ e)\ v \mapsto [v/x]\, e}\ \textbf{(ev-beta)} \qquad \frac{}{\textbf{fix}\ f.\ e \mapsto [\textbf{fix}\ f.\ e\ /\ f]\, e}\ \textbf{(ev-fix)}$$

$$\frac{e \mapsto e'}{\textbf{case}\ e\ \textbf{of}\ \ldots \mapsto \textbf{case}\ e'\ \textbf{of}\ \ldots}\ \textbf{(ev-case-arg)} \qquad \frac{}{\textbf{case}\ c(v)\ \textbf{of}\ \ldots \mid c(x) \Rightarrow e \mid \ldots\ \ \mapsto\ [v/x]\, e}\ \textbf{(ev-case)}$$

Figure 13: Evaluation rules.

1. **Case (contra)**: $\boxed{\mathcal{D} = \dfrac{\varphi, a{:}\gamma \models \bot}{\varphi, a{:}\gamma; \Gamma \vdash e : A}\ \textbf{(contra)}}$

$\varphi \vdash i : \gamma$	Given
$\varphi \models \bot$	By Property 1
$\varphi; [i/a]\Gamma \vdash e : [i/a]A$	By (contra)

2. **Case (sub)**: $\boxed{\mathcal{D} = \dfrac{\varphi, a{:}\gamma; \Gamma \vdash e : B \qquad \varphi, a{:}\gamma \vdash B \trianglelefteq A}{\varphi, a{:}\gamma; \Gamma \vdash e : A}\ \textbf{(sub)}}$

$\varphi; [i/a]\Gamma \vdash e : [i/a]\, B$	By the IH
$\varphi; [i/a]\Gamma \vdash [i/a]\, B \trianglelefteq [i/a]\, A$	By Lemma 1
$\varphi; [i/a]\Gamma \vdash e : [i/a]\, A$	By (sub)

3. **Case (fst)**: $\boxed{\mathcal{D} = \dfrac{\varphi, a{:}\gamma; \Gamma \vdash e : A * B}{\varphi, a{:}\gamma; \Gamma \vdash \pi_1(e) : A}\ \textbf{(fst)}}$

$\varphi; [i/a]\Gamma \vdash e : [i/a]\ A * B$	By the IH
$\varphi; [i/a]\Gamma \vdash e : [i/a]A * [i/a]B$	By definition of substitution
$\varphi; [i/a]\Gamma \vdash \pi_1(e) : [i/a]A$	By (fst)

■

Lemma 14 (Substitution of a program variable)**.** *If* $\varphi; \Gamma, x{:}B \vdash e : A$ *and* $\varphi; \Gamma \vdash v : B$*, then* $\varphi; \Gamma \vdash [v/x]\, e : A$.

Proof. By induction on the derivation $\mathcal{D}$ of $\varphi; \Gamma, x{:}B \vdash e : A$. We show only a few cases.

1. **Case (var)**: $\boxed{\mathcal{D} = \dfrac{(\Gamma, x{:}B)\ (x) = A}{\varphi; \Gamma, x{:}B \vdash x : A}\ \textbf{(var)}}$

   $(\Gamma, x{:}B)\ (x) = A$ so it must be that $A = B$. We are given $\varphi; \Gamma \vdash v : B$, which—since $[v/x]\, x = v$—is the same as $\varphi; \Gamma \vdash [v/x]\, x : A$.

2. **Case (sub)**: By the induction hypothesis, $\varphi; \Gamma \vdash [v/x]\, e : A$. From the derivation, $\varphi \vdash A \trianglelefteq B$. Then by (sub), $\varphi; \Gamma \vdash [v/x]\, e : B$.

3. **Case (let)**: $$\boxed{\mathcal{D} = \dfrac{\varphi; \Gamma, x{:}B \vdash e_1 : B' \qquad \varphi; \Gamma, x{:}B, y{:}B' \vdash e_2 : A}{\varphi; \Gamma, x{:}B \vdash \textbf{let}\ y = e_1\ \textbf{in}\ e_2\ \textbf{end} : A}\ \textbf{(let)}}$$

$\varphi; \Gamma, x{:}B \vdash e_1 : B'$	From $\mathcal{D}$
$\varphi; \Gamma \vdash [v/x]\, e_1 : B'$	By the IH
$\varphi; \Gamma, x{:}B, y{:}B' \vdash e_2 : A$	From $\mathcal{D}$
$\varphi; \Gamma, y{:}B', x{:}B \vdash e_2 : A$	Reordering the context
$\varphi; \Gamma, y{:}B' \vdash [v/x]\, e_2 : A$	By the IH
$\varphi; \Gamma \vdash \textbf{let}\ y = [v/x]\, e_1\ \textbf{in}\ [v/x]\, e_2\ \textbf{end} : A$	By (let)
$\varphi; \Gamma \vdash [v/x]\, \textbf{let}\ y = e_1\ \textbf{in}\ e_2\ \textbf{end} : A$	By definition of substitution

■

Lemma 15. *If $\varphi \models P_1$ and $\varphi, P_1 \vdash A \trianglelefteq B$ then $\varphi \vdash A \trianglelefteq B$.*

Proof. By induction on the derivation of $\varphi, P_1 \vdash A \trianglelefteq B$, and totally straightforward in all but one case:

1. **Case (sort-index-$\trianglelefteq$)**: $$\boxed{\mathcal{D} = \dfrac{\delta_1 \preceq \delta_2 \qquad \varphi, P_1 \models i \doteq j}{\varphi, P_1 \vdash \delta_1(i) \trianglelefteq \delta_2(j)}\ \textbf{(sort-index-}\trianglelefteq\textbf{)}}$$

$\varphi \models P_1$	Given
$\varphi, P_1 \models i \doteq j$	From $\mathcal{D}$
$\varphi \models i \doteq j$	By Property 6
$\delta_1 \preceq \delta_2$	From $\mathcal{D}$
$\varphi \vdash \delta_1(i) \trianglelefteq \delta_2(j)$	By (sort-index-$\trianglelefteq$)

■

Lemma 16. *If $\varphi \models P_1$ and $\mathcal{D}$ derives $\varphi, P_1; \Gamma \vdash e : A$ then $\varphi; \Gamma \vdash e : A$ can be derived.*

Proof. Straightforward (using Lemma 15) in all cases but one:

1. **Case (contra)**: $$\boxed{\mathcal{D} = \dfrac{\varphi, P_1 \models \bot}{\varphi, P_1; \Gamma \vdash e : A}\ \textbf{(contra)}}$$

$\varphi \models P_1$	Given
$\varphi, P_1 \models \bot$	From $\mathcal{D}$
$\varphi \models \bot$	By Property 6
$\varphi; \Gamma \vdash e : A$	By (contra)

■

Lemma 17 (Inversion on $*$)**.** *If $\mathcal{D}$ derives $\vdash v : B$ and $\mathcal{D}'$ derives $\vdash B \trianglelefteq B_1 * B_2$, then v has the form (v_1, v_2) where $\vdash v_1 : B_1$ and $\vdash v_2 : B_2$.*

Proof. By induction on the derivations of the typing and subtyping judgments.

1. **Case (prod)**: $$\boxed{\mathcal{D} = \dfrac{\vdash v_1 : A_1 \qquad \vdash v_2 : A_2}{\vdash (v_1, v_2) : A_1 * A_2}\ \textbf{(prod)}}$$

$\vdash v : A_1 * A_2$	Given
$\vdash A_1 * A_2 \trianglelefteq B_1 * B_2$	Given
$\vdash A_1 \trianglelefteq B_1$	By Lemma 2(*iv*)
$\vdash A_2 \trianglelefteq B_2$	By Lemma 2(*iv*)
$v = (v_1, v_2)$	From $\mathcal{D}$
$\vdash v_1 : A_1$	From $\mathcal{D}$
$\vdash v_2 : A_2$	From $\mathcal{D}$
$\vdash v_1 : B_1$	By (sub)
$\vdash v_2 : B_2$	By (sub)

2. **Case (sect)**: $\boxed{\mathcal{D} = \dfrac{\vdash v : A_1 \qquad \vdash v : A_2}{\vdash v : A_1 \,\&\, A_2}\ \textbf{(sect)}}$

   $\vdash A_1 \,\&\, A_2 \ \trianglelefteq\ B_1 * B_2$ Given

   By Lemma 2(*vi*), at least one of the following is derivable:

   $$\vdash A_1 \ \trianglelefteq\ B_1 * B_2 \qquad\qquad \vdash A_2 \ \trianglelefteq\ B_1 * B_2$$

   If the judgment on the left is derivable, then:

$\vdash A_1 \ \trianglelefteq\ A_1' * A_2'$	By Lemma 7
$\vdash A_1' \ \trianglelefteq\ B_1$	By Lemma 7
$\vdash A_2' \ \trianglelefteq\ B_2$	By Lemma 7
$\vdash v_1 : A_1'$	By the IH
$\vdash v_2 : A_2'$	By the IH
$\vdash v_1 : B_1$	By (sub)
$\vdash v_2 : B_2$	By (sub)

   If the judgment on the left is not derivable, the judgment on the right must be. The next steps are similar, with A_2 in place of A_1.

3. **Case (pi-intro)**: $\boxed{\mathcal{D} = \dfrac{a{:}\gamma \vdash v : A}{\vdash v : \Pi a{:}\gamma.\, A}\ \textbf{(pi-intro)}}$

$\vdash \Pi a{:}\gamma.\, A \ \trianglelefteq\ B_1 * B_2$	Given
$\vdash [i/a]\, A \ \trianglelefteq\ B_1 * B_2$	(for some i s.t. $\vdash i : \gamma$) Inversion: (pi-1-$\trianglelefteq$) was used
$\vdash v : [i/a]\, A$	By substitution (Lemma 13)

   By the IH applied to the last line, $v = (v_1, v_2)$ and $\vdash v_1 : B_1$ and $\vdash v_2 : B_2$.

4. **Case (sub)**: By transitivity, $\vdash A \ \trianglelefteq\ B_1 * B_2$. By Lemma 7, there exist A_1, A_2 such that $\vdash A \ \trianglelefteq\ A_1 * A_2$ and $\vdash A_1 \ \trianglelefteq\ B_1$ and $\vdash A_2 \ \trianglelefteq\ B_2$. By the IH, $v = (v_1, v_2)$ and $\vdash v_1 : A_1$ and $\vdash v_2 : A_2$.

   Applying (sub) twice, we obtain $\vdash v_1 : B_1$ and $\vdash v_2 : B_2$.

5. **Case (contra)**: The context φ is empty. By Property 4, (contra) could not have been applied.

■

Lemma 18 (Variable Type Shrinking)**.** *If $\mathcal{D}$ is a derivation of $\varphi; \Gamma, x{:}A \vdash e : B$ and $\varphi \vdash A' \ \trianglelefteq\ A$, then $\varphi; \Gamma, x{:}A' \vdash e : B$ is derivable.*

Proof. By induction on $\mathcal{D}$. Essentially, in $\mathcal{D}$: (1) replace $\Gamma, x{:}A$ with $\Gamma, x{:}A'$, and (2) replace every use of (var-up) on x with a (var-up) and a (sub):

$$\dfrac{\dfrac{(\Gamma, x{:}A')(x) = A'}{\varphi; \Gamma, x{:}A' \vdash x : A'} \qquad \varphi \vdash A' \ \trianglelefteq\ A}{\varphi; \Gamma, x{:}A' \vdash x : A}$$

This is legitimate because (var-up) is the only rule that inspects Γ.

$\mathcal{D}$ derived $\varphi; \Gamma, x{:}A \vdash e : B$ and we replaced all the $\Gamma, x{:}A$'s with $\Gamma, x{:}A'$'s, so we have the desired derivation of $\varphi; \Gamma, x{:}A' \vdash e : B$. ■

Lemma 19 (Inversion on $\rightarrow$)**.** *If $\mathcal{D}$ is a derivation of $\vdash v : B$ and $\vdash B \ \trianglelefteq\ B_1 \rightarrow B_2$, then v has the form **lam** $x.\, e$ where $x{:}B_1 \vdash e : B_2$.*

Proof. By cases on the rule concluding the derivation $\mathcal{D}$.

1. **Case (sect)**: Similar to the case in the proof of Lemma 17.

2. **Case (pi-intro)**: Similar to the case in the proof of Lemma 17.

3. **Case (lam)**: $\boxed{\mathcal{D} = \dfrac{\cdot; x{:}A_1 \vdash e : A_2}{\vdash \textbf{lam}\; x.\, e : A_1 \to A_2}\ \textbf{(lam)}}$

$\vdash A_1 \to A_2 \trianglelefteq B_1 \to B_2$	Given
$\vdash B_1 \trianglelefteq A_1$	By Lemma 2(*iii*)
$\cdot; x{:}A_1 \vdash e : A_2$	From $\mathcal{D}$
$\cdot; x{:}B_1 \vdash e : A_2$	By Lemma 18
$\vdash A_2 \trianglelefteq B_2$	By Lemma 2(*iii*)
$\cdot; x{:}B_1 \vdash e : B_2$	By (sub)

4. **Case (sub)**: $\boxed{\mathcal{D} = \dfrac{\vdash e : A \qquad \vdash A \trianglelefteq B}{\vdash e : B}\ \textbf{(sub)}}$

$\vdash A \trianglelefteq B$	From $\mathcal{D}$
$\vdash B \trianglelefteq B_1 \to B_2$	Given
$\vdash A \trianglelefteq B_1 \to B_2$	By transitivity (Lemma 8)
$v = \textbf{lam}\; x.\, e$	By the IH
$x{:}B_1 \vdash e : B_2$	By the IH

5. **Case (contra)**: By Property 4, (contra) could not have been used.

■

Lemma 20 (Inversion on $\delta(i)$)**.** *If $\mathcal{D}$ is a derivation of $\vdash v : B$ and $\vdash B \trianglelefteq \delta(i)$, then there exist v', c, δ_2, j such that $v = c(v')$, $\vdash v' : A$, $c : A \to \delta_2(j)$, and $\vdash \delta_2(j) \trianglelefteq \delta(i)$.*

Proof. By induction on the derivation $\mathcal{D}$ of $\vdash v : B$. Cases:

1. **Case (sub)**: By transitivity of subtyping (Lemma 8), we can apply the IH, yielding the result.

2. **Case (contra)**: Could not have been used (Property 4).

3. **Case (pi-intro)**: $\boxed{\mathcal{D} = \dfrac{a{:}\gamma \vdash v : A}{\vdash v : \Pi a{:}\gamma.\, A}\ \textbf{(pi-intro)}}$

   By Lemma 2(*v*) we have
   $$\vdash [j/a]\, A \trianglelefteq \delta(i)$$
   for some j such that $\vdash j : \gamma$. Replacing a with j in the derivation of $a{:}\gamma \vdash v : A$ yields an equally long derivation of $\vdash v : [j/b]\, A$ to which we can apply the induction hypothesis.

4. **Case (sect)**: If $B = A_1 \,\&\, A_2 \trianglelefteq \delta(i)$ then at least one of $\vdash A_1 \trianglelefteq \delta(i)$ and $\vdash A_2 \trianglelefteq \delta(i)$ is derivable (Lemma 2(*vi*)). Thus we can apply the IH to yield the result.

5. **Case (cons-app)**: $\boxed{\mathcal{D} = \dfrac{\vdash c : A \to \delta_2(j_2) \qquad \vdash \delta_2(j_2) \trianglelefteq \delta_1(j_1) \qquad \vdash v' : A}{\vdash c(v') : \delta_1(j_1)}\ \textbf{(cons-app)}}$

   The rule types a term v of the form $c(e)$. By the definition of values, e must be some value v'.

$\vdash \delta_2(j_2) \trianglelefteq \delta_1(j_1)$	From derivation $\mathcal{D}$
$\vdash \delta_1(j_1) \trianglelefteq \delta(i)$	Given
$\vdash \delta_2(j_2) \trianglelefteq \delta(i)$	By Lemma 8 (transitivity of subtyping)
$\vdash c : A \to \delta_2(j_2)$	From derivation $\mathcal{D}$
$\vdash v' : A$	From derivation $\mathcal{D}$

   The last three judgments constitute the result.

■

Lemma 21 (Constructor Typing Inversion)**.** *If $\vdash c : A \to \delta'(j')$ such that $\vdash \delta'(j') \trianglelefteq \delta(j)$, then there exist a natural number k and an index i (where $\vdash i : \gamma_{\delta k}$) such that $A = [i/a_{\delta k}]\, L_{\delta k}$ and $\models j \doteq [i/a_{\delta k}]\, j_{\delta k}$.*

Proof. There is only one rule for typing a constructor, (cons), so the derivation was

$$\frac{\mathcal{S}(c) = \big(\Pi a_1{:}\gamma_1.\, L_1 \to \delta_1(j_1)\big) \;\&\; \cdots \;\&\; \big(\Pi a_n{:}\gamma_n.\, L_n \to \delta_n(j_n)\big) \qquad \varphi \vdash i : \gamma_\ell}{\varphi \vdash c : [i/a_\ell]\;\; L_\ell \to \delta_\ell(j_\ell)}$$

with $\delta_\ell = \delta' \preceq \delta$. By definition of $A_{\delta k}$, if $\Pi a_\ell{:}\gamma_\ell.\, L_\ell \to \delta_\ell(j)$ is a conjunct of $\mathcal{S}(c)$ and δ_ℓ is a subsort of δ, that conjunct is among the $A_{\delta k}$'s. Therefore, there is a k such that

$$A = [i/a_{\delta k}]L_{\delta k}, \qquad j' = [i/a_{\delta k}]\, j_{\delta k}.$$

By inversion it follows from $\vdash \delta'(j') \;\trianglelefteq\; \delta(j)$ that $\models\, j' \doteq j$, so by the identity of j' and $[i/a_{\delta k}]\, j_{\delta k}$,

$$\models\, j \doteq [i/a_{\delta k}]\, j_k$$

■

3.7 Type safety

We state and prove type safety as a single theorem, rather than as two separate theorems of preservation (stepping does not change types) and progress (every term either steps or is a value). The proofs of the two separate theorems would share most of their structure, so it is easier to prove the combined theorem.

Theorem 22 (Type Safety)**.** *If* $\cdot;\cdot \vdash e : A$*, then either*

1. *e is a value, or*
2. *there exists a unique term e' such that $e \mapsto e'$ and $\cdot;\cdot \vdash e' : A$.*

Proof. By induction on the derivation $\mathcal{D}$ of $\vdash e : A$. We need no cases for the rules excluded from the undirected system: (anno-up), (sect-up-. . .), and (pi-elim). Suppose e is not a value (if it is a value, we're done). Now we can dispense with cases for those rules that can only type values: (sect), (pi-intro), (var), (unit), and (lam). We will show that there exists an e' with the desired properties; we know by Theorem 12 that e' must be unique.

1. **Case (sub)**: $$\boxed{\mathcal{D} = \frac{\vdash e : A \qquad \vdash A \trianglelefteq B}{\vdash e : B}\ \textbf{(sub)}}$$

$\vdash e : A$	From the derivation $\mathcal{D}$
$e \mapsto e'$	By induction hypothesis
$\vdash e' : A$	By induction hypothesis
$\vdash A \trianglelefteq B$	From $\mathcal{D}$
$\vdash e' : B$	By (sub)

2. **Case (contra)**: By Property 4, the (contra) rule could not have derived $\cdot;\cdot \vdash e : A$.

3. **Case (app)**: $$\boxed{\mathcal{D} = \frac{\vdash e_1 : B \to A \qquad \vdash e_2 : B}{\vdash e_1(e_2) : A}\ \textbf{(app)}}$$

   There are three cases, depending on which of e_1 and e_2 are values.

   - e_1 and e_2 are values. By Lemma 19 the applied expression e_1 has the form **lam** $x.\, e$ and $x{:}B \vdash e : A$. Thus, $e_1(e_2) \mapsto [e_2/x]e$ (ev-beta), and by Lemma 14, $\vdash [e_2/x]e : A$.
   - e_1 is a value but e_2 is not a value: By the IH, $e_2 \mapsto e_2'$ and $\vdash e_2' : A$. So by (ev-app-arg) $e_1(e_2)$ steps to $e_1(e_2')$. By (app), $\vdash e_1(e_2') : B$.
   - e_1 is not a value: $e_1 \mapsto e_1'$ (ev-app-fn) and $\vdash e_1' : B \to A$. Therefore $e_1(e_2) \mapsto e_1'(e_2)$, which by (app) has type A.

4. **Case (let)**: $$\boxed{\mathcal{D} = \frac{\cdot;\cdot \vdash e_1 : B \qquad \cdot;x{:}B \vdash e_2 : A}{\cdot;\cdot \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} : A}\ \textbf{(let)}}$$

- Suppose e_1 is a value.

$$\begin{array}{rll} \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} & \mapsto [e_1/x]\, e_2 & \text{By (ev-let)} \\ & \vdash [e_1/x]\, e_2 : A & \text{By Lemma 14} \end{array}$$

- Suppose e_1 is not a value.

$$\begin{array}{rll} & \vdash e_1 : B & \text{From } \mathcal{D} \\ e_1 & \mapsto e_1' & \text{By the IH} \\ & \vdash e_1' : B & \text{By the IH} \\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end} & \mapsto \textbf{let}\ x = e_1'\ \textbf{in}\ e_2\ \textbf{end} & \text{By (ev-let-arg)} \\ & \vdash \textbf{let}\ x = e_1'\ \textbf{in}\ e_2\ \textbf{end} : A & \text{By (let)} \end{array}$$

5. **Case (fix)**: $\boxed{\mathcal{D} = \dfrac{\cdot; f{:}A \vdash e : A}{\cdot;\cdot \vdash \textbf{fix}\ f.\, e : A}\ \textbf{(fix)}}$

$$\begin{array}{ll} \textbf{fix}\ f.\, e \mapsto [\textbf{fix}\ f.\, e/f]\, e & \text{By (ev-fix)} \\ \cdot; f{:}A \vdash e : A & \text{From } \mathcal{D} \\ \cdot;\cdot \vdash [\textbf{fix}\ f.\, e/f]\, e : A & \text{Substitution (14)} \end{array}$$

6. **Case (fst)**: $\boxed{\mathcal{D} = \dfrac{\vdash e : A * B}{\vdash \pi_1(e) : A}\ \textbf{(fst)}}$

By the induction hypothesis, either e is a value or $e \mapsto e'$ with $\vdash e' : A * B$.

- If e is a value, by inversion (Lemma 17) it must be some (v_1, v_2) with $\vdash v_1 : A$ and $\vdash v_2 : B$. By (ev-fst), $\pi_1(e) \mapsto v_1$. Since $\vdash v_1 : A$, we're done.
- If $e \mapsto e'$ then $\pi_1(e) \mapsto \pi_1(e')$ by (ev-fst-arg). By the IH, $\vdash e' : A{*}B$. By (fst), $\vdash \pi_1(e') : A$.

7. **Case (snd)**: Similar to (fst).

8. **Case (prod)**: $e = (e_1, e_2)$.

- Suppose e_1 is not a value.

$$\begin{array}{rll} e_1 & \mapsto e_1' & \text{By the IH} \\ & \vdash e_1' : A & \text{By the IH} \\ (e_1, e_2) & \mapsto (e_1', e_2) & \text{By (ev-pair-1)} \\ & \vdash (e_1, e_2) : A * B & \text{By (prod)} \end{array}$$

- If e_1 is a value, then e_2 must not be a value (otherwise (e_1, e_2) would itself be a value, contrary to assumption).

$$\begin{array}{rll} e_2 & \mapsto e_2' & \text{By the IH} \\ & \vdash e_2' : B & \text{By the IH} \\ (e_1, e_2) & \mapsto (e_1, e_2') & \text{By (ev-pair-2)} \\ & \vdash (e_1, e_2') : A * B & \text{By (prod)} \end{array}$$

9. **Case (cons-app)**: If e is a value, then $c(e)$ is a value and there is no more to do. If e is not a value, by the IH, $e \mapsto e'$ and $\vdash e' : A$. Applying (cons-app) yields $\vdash c(e') : \delta_1(j)$.

10. **Case (case)**: $\boxed{\mathcal{D} = \dfrac{\vdash e : \delta(i) \qquad \overset{\mathcal{D}'}{\vdash ms \downarrow_{\delta(i)} C}}{\vdash \textbf{case}\ e\ \textbf{of}\ ms\ :\ C}\ \textbf{(case)}}$

If e is not a value, then by the IH, $e \mapsto e'$ and $e' : \delta(i)$.

If e is a value, we step by (ev-case). By our requirement that **case** expressions be exhaustive, ms contains a match $c(x) \Rightarrow e'$ for every constructor c, so there must be a subderivation

$$a_{\delta k}{:}\gamma_{\delta k}, i \doteq j_{\delta k}; x{:}L_{\delta k} \vdash e' : C$$

contained in $\mathcal{D}'$.

$$\begin{array}{rl}
\vdash e : \delta(i) & \text{From } \mathcal{D} \\
\exists v.\ e = c(v) & \text{By Lemma 20} \\
\vdash c : A \to \delta_k(j) & \text{By Lemma 20} \\
\vdash \delta_k(j) \trianglelefteq \delta(i) & \text{By Lemma 20} \\
\exists k \in \mathbf{N}, j'.\ \vdash j' : \gamma_{\delta k} & \text{By Lemma 21} \\
\text{and } [j'/a_{\delta k}]\, L_{\delta k} = A & \\
a_{\delta k}{:}\gamma_{\delta k}, i \doteq j_{\delta k}; x{:}L_{\delta k} \vdash e' : C & \text{From } \mathcal{D}' \\
\cdot, i \doteq [j'/a_{\delta k}] j_{\delta k}; x{:}A \vdash e' : [j'/a_{\delta k}]\, C & \text{Substituting } j' \text{ for } a_{\delta k} \\
\cdot, i \doteq [j'/a_{\delta k}] j_{\delta k}; x{:}A \vdash e' : C & a_{\delta k} \text{ not free in } C \\
& \\
\models i \doteq [j'/a_{\delta k}]\, j_{\delta k} & \text{By Lemma 21} \\
\cdot; x{:}A \vdash e' : C & \text{Lemma 16} \\
& \\
\vdash v : A & \text{By Lemma 20} \\
\vdash [v/x]\, e' : C & \text{By substitution (Lemma 14)}
\end{array}$$

■

Remark 3. We showed above that if (pi-intro) is not restricted to typing values, the system becomes unsound. Here is how the proof fails without a value restriction: We would need to apply the induction hypothesis to a premise $a{:}\gamma; \cdot \vdash e : A$, which has a φ context that is not empty! We see no way to generalize the hypothesis: it is unclear what evaluating under any φ but $\cdot$ means, especially if γ is an empty sort such as $\perp$.

4 Example: red-black trees

A *red-black tree* is a classic form of binary search tree. Each node in a red-black tree is either empty or a branch node; a branch node is either red or black. An empty node is considered to be black. Two invariants ensure that red-black trees are (approximately) balanced:

(1) the children of every red node are black;

(2) for every node x, there exists a natural number $bh(x)$ such that the number of black nodes on every path from x to its leaves is $bh(x)$.

We call $bh(x)$ the *black height* of x. The useful consequence of the two invariants is that the height of a red-black tree containing n non-empty nodes can be at most $2\log_2(n+1)$.

The obvious algebraic datatype, in SML syntax, is

```
datatype 'a dict = E                              (* empty *)
                 | R of 'a * 'a dict * 'a dict  (* red branch *)
                 | B of 'a * 'a dict * 'a dict  (* black branch *)
```

However, for simplicity—and because our language lacks polymorphism—we will refine this datatype instead:

```
datatype dict = E                 (* empty *)
              | R of dict * dict  (* red branch *)
              | B of dict * dict  (* black branch *)
```

Our type system can express both invariants. Datasort refinements alone are sufficient to express the color invariant (1), but not the black height invariant (2), since a node's black height belongs to an infinite set (the natural numbers); we cannot express the fact that the black heights of the left and right children are the same element of that set with only a finite collection of datasorts. But we *can* use index refinements for this purpose. Indexing a tree by the number of nodes it contains and the tree's black height leads to the constructor types of Figurereffig:rbt-sig. Invariant (2) is captured simply by writing h for the second index of both arguments, in every conjunct. We do not represent a third invariant that for any branch node x containing a key k, the keys in its left subtree are less than k and the keys in its right subtree are greater than k.

Our subsort relation (Figure 14) is more elaborate than one might expect. This is because in practice, it is expedient to temporarily break the color invariant (1): the operations on red-black trees are typically implemented using functions that take a tree in which either the root or a child of the root may not satisfy the color invariant and do appropriate rotations, returning a tree satisfying both invariants.[4] If a tree has sort $\mathsf{badRoot}$, the color invariant *may* be violated at the root; if it has sort $\mathsf{badLeft}$ (or $\mathsf{badRight}$), the color invariant *may* be violated at the left (or right) child. Sort rbt is the sort of trees for which no invariant is violated, and red and black are simply the sorts of trees with a root of that color satisfying both invariants.[5]

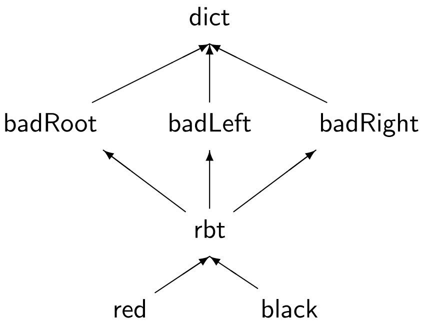

Figure 14: The subsort relation for red-black trees.

Figure 16 shows the (match-δ-c) inference rules generated by the method described in Section 3.3. (As an aside, some of the rules are suboptimal in that some of the premises are redundant; $\mathsf{rbt}(\dots) * \mathsf{rbt}(\dots)$, the type of y in the last premise of (match-dict-black), is a supertype of the types of y in the two premises that precede it.)

4.1 Typechecking `restore_right`

To see how the rules work on real code, consider the `restore_right` function in Figure 17. Given a red-black tree t with a possible color violation at t's right child (i.e. t's right child may be red but has a non-black child), `restore_right` returns an equivalent tree with no color violations. We express this behavior through a type annotation:

$$(\textbf{lam}\ arg.\ \dots) : \Pi n{:}\mathcal{N}.\, \Pi h{:}\mathcal{N}.\, \mathsf{badRight}(n,h) \to \mathsf{rbt}(n,h)$$

We do not index the tree by the keys it contains, so we cannot check that the result actually contains the same keys. However, we can and do use the first index refinement to check that the result has the same number of nodes n as the argument.

Since our language has only non-nested and tuple-free patterns, a full translation of `restore_right` would be tedious and confusing. Instead, we show three of the cases (Figure 18). Typechecking the function requires that, for each case arm, we derive a judgment

$$arg{:}\mathsf{badRight}(n,h), \dots ; n{:}\mathcal{N}, h{:}\mathcal{N}, \dots \vdash \textit{case-arm} : \mathsf{rbt}(n,h)$$

1. In the first case, the argument is just `E`, and the body of the case arm is `E` as well. We need to satisfy the premise of (match-badRoot-`E`),

$$arg{:}\mathsf{badRight}(n,h); n{:}\mathcal{N}, h{:}\mathcal{N}, (n,h) \doteq (0,0) \vdash \mathtt{E} \downarrow \mathsf{rbt}(n,h)$$

   The derivation is simple:

$$\begin{array}{rll}
\dots;\dots \vdash \mathtt{E} \uparrow \mathsf{black}(0,0) & & \text{By (cons-app)} \\
\mathsf{black} \preceq \mathsf{rbt} & & \\
n{:}\mathcal{N}, h{:}\mathcal{N}, (n,h) \doteq (0,0) \models (0,0) \doteq (n,h) & & \\
n{:}\mathcal{N}, h{:}\mathcal{N}, (n,h) \doteq (0,0) \vdash \mathsf{black}(0,0) & \trianglelefteq\ \mathsf{rbt}(n,h) & \text{By (sort-index-}\trianglelefteq\text{)} \\
arg{:}\mathsf{badRight}(n,h); n{:}\mathcal{N}, h{:}\mathcal{N}, (n,h) \doteq (0,0) \vdash \mathtt{E} \downarrow \mathsf{rbt}(n,h) & & \text{By (sub-down)}
\end{array}$$

[4] See, for instance, Okasaki's Standard ML implementation of red-black trees [15].

[5] This refinement scheme for red-black trees is due to Rowan Davies.

$$\begin{aligned}
\mathcal{S}(\mathtt{E}) &= \Pi a{:}\mathbf{1}.\,\mathbf{1} \to \mathsf{black}(0,0),\\
\mathcal{S}(\mathtt{R}) &= \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{black}(n_L,h) * \mathsf{black}(n_R,h) \to \mathsf{red}(n_L + n_R + 1, h)\big)\\
&\quad \&\ \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{black}(n_L,h) * \mathsf{rbt}(n_R,h) \to \mathsf{badRoot}(n_L + n_R + 1, h)\big)\\
&\quad \&\ \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{rbt}(n_L,h) * \mathsf{black}(n_R,h) \to \mathsf{badRoot}(n_L + n_R + 1, h)\big),\\
\mathcal{S}(\mathtt{B}) &= \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{badRoot}(n_L,h) * \mathsf{rbt}(n_R,h) \to \mathsf{badLeft}(n_L + n_R + 1, h+1)\big)\\
&\quad \&\ \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{rbt}(n_L,h) * \mathsf{badRoot}(n_R,h) \to \mathsf{badRight}(n_L + n_R + 1, h+1)\big)\\
&\quad \&\ \big(\Pi n_L, h, n_R{:}\mathcal{N}.\, \mathsf{rbt}(n_L,h) * \mathsf{rbt}(n_R,h) \to \mathsf{black}(n_L + n_R + 1, h+1)
\end{aligned}$$

Figure 15: Signature of red-black tree constructors. For clarity, we write $\Pi n_L, h, n_R{:}\mathcal{N}.\ \ldots n_L \ldots h \ldots n_R$ instead of $\Pi a{:}\mathcal{N} * (\mathcal{N} {*} \mathcal{N}).\ \ldots \mathrm{fst}(a) \ldots \mathrm{fst}(\mathrm{snd}(a)) \ldots \mathrm{snd}(\mathrm{snd}(a))$, which is what our formulation actually requires.

2. In the second case, the argument has the form $\mathtt{B}(\mathtt{R}(ll, lr), \mathtt{R}(rl, rr))$. When we check the case arm $\mathtt{R}(\mathtt{B}(ll, lr), \mathtt{B}(rl, rr))$ against $\mathsf{rbt}(n, h)$, the contexts φ and Γ are as follows:

$$\begin{aligned}
\varphi = {} & n{:}\mathcal{N}, h{:}\mathcal{N}, a_L{:}\mathcal{N}, a_R{:}\mathcal{N}, b{:}\mathcal{N},\\
& (n,h) \doteq (a_L + a_R + 1, b+1),\\
& a_{LL}{:}\mathcal{N}, a_{LR}{:}\mathcal{N}, a_L \doteq a_{LL} + a_{LR} + 1,\\
& a_{RL}{:}\mathcal{N}, a_{RR}{:}\mathcal{N}, a_R \doteq a_{RL} + a_{RR} + 1;\\
\Gamma = {} & arg{:}\mathsf{badRight}(n,h),\\
& l{:}\mathsf{rbt}(a_L, b),\\
& r{:}\mathsf{badRoot}(a_R, b),\\
& ll{:}\mathsf{black}(a_{LL}, b), lr{:}\mathsf{black}(a_{LR}, b),\\
& rl{:}\mathsf{rbt}(a_{RL}, b), rr{:}\mathsf{black}(a_{RR}, b)
\end{aligned}$$

Here is an outline of the derivation.

$\varphi; \Gamma \vdash \mathtt{B}(ll, lr) \downarrow \mathsf{black}(a_{LL}{+}a_{LR}{+}1, b{+}1)$	By (cons) and (cons-app)
$\varphi; \Gamma \vdash \mathtt{B}(rl, rr) \downarrow \mathsf{black}(a_{RL}{+}a_{RR}{+}1, b{+}1)$	By (cons) and (cons-app)
$\varphi \models (n,h) \doteq ((a_{LL}{+}a_{LR}{+}1) + (a_{RL}{+}a_{RR}{+}1) + 1, b{+}1)$	See φ above
$\varphi \vdash \mathsf{red}(a_{LL}{+}a_{LR}{+}1) + (a_{RL}{+}a_{RR}{+}1) + 1, b{+}1) \ \trianglelefteq\ \mathsf{rbt}(n,h)$	By (sort-index-$\leq$)
$\varphi; \Gamma \vdash \mathtt{R}(\mathtt{B}(ll, lr), \mathtt{B}(rl, rr)) \downarrow \mathsf{rbt}(n,h)$	By (cons) and (cons-app)

3. In the third case, the argument matches $\mathtt{B}(\mathtt{B}(ll, lr), \mathtt{R}(\mathtt{R}(rll, rlr), rr))$. The derivation proceeds much as in the preceding case, so we omit it.

4.2 Comparison to the individual refinement systems

There is no way to express the black height invariant with datasort refinements, so the combined system has a clear advantage over datasort refinements alone. The advantage over index refinements is not as obvious, for Xi showed that it is possible to check the color invariant using index refinements; one simply encodes the color as an integer in $\{0, 1\}$ and refines the red-black tree datatype by a tuple whose first component is the encoded color [18]. However, we consider this encoding both inelegant and unfortunate: the type annotations needed are substantially less clear than ours. For users, the burdens of using type refinements are precisely inventing appropriate refinements and writing refined type annotations, so these burdens should be as light as possible.

Unfortunately, the `insert` function which inserts a key k into a tree x cannot be typed in our system. The height of `insert`'s result is not uniquely determined by the height and number of elements of x, so we have nothing to write for the index of `insert`'s result. To type `insert`, we need existential dependent types.

5 Conclusion

We have presented a language and type system combining two kinds of type refinements: datasort refinements and index refinements, proved its soundness, and shown how typechecking works in a small realistic example. However, we have not included several important features. Our system lacks polymorphism and effects, but it seems likely that the approaches taken by Davies and Pfenning [7] for datasort refinements alone are directly applicable. We already have a value restriction in the introduction rules for both intersection and dependent universal types (the (sect-down) and (pi-intro) rules, Figure 9).

So far, we have followed Xi in constructing a system parametric in some constraint domain, but presenting only examples in the domain of integers. Other domains hold the hope of expressing still more properties. Our system can express the invariants of red-black trees (color and black height), but cannot express all the properties of the basic operations—`insert`, etc.—on red-black trees; it can guarantee that the result of `restore_right` satisfies the invariants, but it cannot check that `restore_right` returns a tree containing the same information (keys) as the original. But suppose we indexed red-black trees by a set of keys. A system capable of doing this would *perfectly* refine red-black trees: any two red-black trees with the same refinements (datasort and index) would have to be *the same* insofar as having the same contents and the same behavior in terms of the basic operations.[6] Whether typechecking in such domains is feasible is an open question; tools such as Hilberticus[13], which implements a decision procedure for a fragment of set theory, indicate there is more hope than one might suppose.

We have tried to design our system to be as simple as possible. In particular, our system of types, while very much in the spirit of Xi and Pfenning's original work [17], does not include several of its constructs, notably subset sorts $\{a{:}s \mid P\}$ (for example, the sort of integers greater than 2 is $\{a{:}\mathcal{N} \mid a > 2\}$) and various propositional connectives. These constructs are entirely compatible with the system—we simply put them in place of the ellipses in Figure 2. (One might worry that issues would arise from empty subset sorts in combination with the contradiction rule, but we already have an empty sort $\bot$). However, logical conjunction $\wedge$ may be redundant in the presence of intersection types: it would seem that, instead of

$$\Pi a{:}\{a{:}s \mid P \wedge Q\}.\, A$$

we could write

$$(\Pi a{:}\{a{:}s \mid P\}.\, A) \,\&\, (\Pi a{:}\{a{:}s \mid Q\}.\, A)$$

But we have not proved this, and at any rate the former type is more legible. Similarly, we might add union types instead of adding logical disjunction $\vee$.

It is natural to question the necessity of the contradiction rule. The system is sound without it, but it is worthwhile in practice since it allows us to exclude many unreachable case arms. As pointed out in Section 3.3, it does require that we do simple typechecking first to avoid admitting nonsensical programs. But doing simple typechecking—essentially *forcing* typechecking to be conservative—allows us to dispense with a restriction [6] on the intersection type operator, simplifying the formal system. (We have not explored applications of the unrestricted use of $\&$. If it turns out to be sufficiently valuable, perhaps a non-conservative extension would be acceptable. Note that one obvious unrestricted use of $\&$, to simulate polymorphism (for example, the identity function can be given the type $(A \to A) \,\&\, (B \to B) \,\&\, \ldots$ for all types $A, B, \ldots$), is subsumed by Hindley-Milner polymorphism.)

Given a solver for some constraint domain, it should be easy to construct a bidirectional typechecker for the language we have presented. However, such a checker would be of limited utility: without existential dependent types (dependent sums), we cannot express the behavior of many functions, such as the `filter` function on lists indexed by their length and the `insert` function on red-black trees. Extending the system with existential dependent types *and* formulating a practical approach to typechecking that system is an important future research direction.

We expect that other classic data structures besides red-black trees could be usefully refined in our system (even without existential dependent types). For example, B-trees seem amenable to refinement: the invariants of a B-tree involve bounds on the number of keys stored at a node—the bounds vary depending on whether the node is the root—and the distance from the root to the leaves. A datasort refinement is suitable for distinguishing roots from internal non-root nodes, and index refinements appear suitable for expressing bounds on the number of keys and the distance to the leaves. Moreover, B-trees are widely used in databases and would constitute a quite compelling application.

[6] We ignore differences in time and space behavior.

Acknowledgments. I would like to thank Frank Pfenning for many enlightening discussions about this work, and Karl Crary, Margaret DeLap, and Derek Dreyer for useful comments on various drafts.

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *21st ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 163–173, 1994.

[2] Lennart Augustsson. Cayenne—a language with dependent types. In *Int'l Conf. Functional Programming (ICFP '98)*, pages 239–250, 1998.

[3] Robert Cartwright and Mike Fagan. Soft typing. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991.

[4] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.

[5] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *J. Combinatorial Theory (A)*, 14:288–297, 1973.

[6] Rowan Davies. Practical refinement-type checking. PhD thesis proposal, Carnegie Mellon University, 1997.

[7] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000.

[8] Ewen Denney. Refining refinement types. In *Informal Proc. Types Workshop on Subtyping, Inheritance and Modular Development of Proofs*, University of Durham; `http://www.dai.ed.ac.uk/daidb/people/homes/ewd/papers/durham.ps`, September 1997.

[9] Tim Freeman. *Refinement types for ML*. PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-110.

[10] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, pages 268–277. ACM Press, 1991.

[11] Susumu Hayashi. Singleton, union, and intersection types for program extraction. *Information and Computation*, 109:174–210, 1994.

[12] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. 1991.

[13] Jörg Lücke. Hilberticus – a tool for deciding an elementary sublanguage of set theory. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Int'l Joint Conference on Automated Reasoning (IJCAR 2001), Siena, Italy, June 18-23, 2001*, volume 2083 of *LNCS*. Springer, 2001.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[15] Chris Okasaki. *Purely Functional Data Structures*. Cambridge, 1998.

[16] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[17] Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.

[18] Hongwei Xi. Dependently typed data structures. Revised version superseding that presented at WAAAPL '99; `http://www.cs.bu.edu/~hwxi/academic/papers/DTDS.pdf`, February 2000.

[19] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symp. Principles of Programming Languages (POPL '99)*, pages 214–227, 1999.

Let φ' be $\varphi, n_L{:}\mathcal{N}, b{:}\mathcal{N}, a_R{:}\mathcal{N}$.

$$\frac{\begin{array}{l}\varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{rbt}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{dict}(i)} C}\;(\textbf{match-}\mathsf{dict}\textbf{-}\mathtt{R})$$

$$\frac{\begin{array}{l}\varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{badRoot}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{badRoot}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{dict}(i)} C}\;(\textbf{match-}\mathsf{dict}\textbf{-}\mathtt{B})$$

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{dict}(i)} C}\;(\textbf{match-}\mathsf{dict}\textbf{-}\mathtt{E}) \quad \frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{badRoot}(i)} C}\;(\textbf{match-}\mathsf{badRoot}\textbf{-}\mathtt{E})$$

$$\frac{\begin{array}{l}\varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{rbt}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{badRoot}(i)} C}\;(\textbf{match-}\mathsf{badRoot}\textbf{-}\mathtt{R})$$

$$\frac{\varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{badRoot}(i)} C}\;(\textbf{match-}\mathsf{badRoot}\textbf{-}\mathtt{B})$$

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{badLeft}(i)} C}\;(\textbf{match-}\mathsf{badLeft}\textbf{-}\mathtt{E}) \quad \frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{badRight}(i)} C}\;(\textbf{match-}\mathsf{badRight}\textbf{-}\mathtt{E})$$

$$\frac{\varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{badLeft}(i)} C}\;(\textbf{match-}\mathsf{badLeft}\textbf{-}\mathtt{R})$$

$$\frac{\begin{array}{l}\varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{badRoot}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{badLeft}(i)} C}\;(\textbf{match-}\mathsf{badLeft}\textbf{-}\mathtt{B})$$

$$\frac{\varphi', i \doteq (a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{badRight}(i)} C}\;(\textbf{match-}\mathsf{badRight}\textbf{-}\mathtt{R})$$

$$\frac{\begin{array}{l}\varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{badRoot}(a_R, b) \vdash e \downarrow C\\ \varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{badRight}(i)} C}\;(\textbf{match-}\mathsf{badRight}\textbf{-}\mathtt{B})$$

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{rbt}(i)} C}\;(\textbf{match-}\mathsf{rbt}\textbf{-}\mathtt{E}) \quad \frac{\begin{array}{c}\varphi', i \doteq (a_L{+}a_R{+}1, b);\\ \Gamma, x{:}\mathsf{black}(a_L, b) * \mathsf{black}(a_R, b) \vdash e \downarrow C\end{array}}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{rbt}(i)} C}\;(\textbf{match-}\mathsf{rbt}\textbf{-}\mathtt{R})$$

$$\frac{\varphi', i \doteq (a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b) * \mathsf{rbt}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{rbt}(i)} C}\;(\textbf{match-}\mathsf{rbt}\textbf{-}\mathtt{B})$$

$$\frac{}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{red}(i)} C}\;(\textbf{match-}\mathsf{red}\textbf{-}\mathtt{E}) \quad \frac{\varphi', i{\doteq}(a_L{+}a_R{+}1, b); \Gamma, x{:}\mathsf{black}(a_L, b){*}\mathsf{black}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{red}(i)} C}\;(\textbf{match-}\mathsf{red}\textbf{-}\mathtt{R})$$

$$\frac{}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{red}(i)} C}\;(\textbf{match-}\mathsf{red}\textbf{-}\mathtt{B}) \quad \frac{\varphi', i{\doteq}(a_L{+}a_R{+}1, b{+}1); \Gamma, y{:}\mathsf{rbt}(a_L, b){*}\mathsf{rbt}(a_R, b) \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{B}(y) \Rightarrow e \downarrow_{\mathsf{black}(i)} C}\;(\textbf{match-}\mathsf{black}\textbf{-}\mathtt{B})$$

$$\frac{\varphi, a{:}\mathbf{1}, i \doteq (0,0); \Gamma, x{:}\mathbf{1} \vdash e \downarrow C}{\varphi; \Gamma \vdash \mathtt{E} \Rightarrow e \downarrow_{\mathsf{black}(i)} C}\;(\textbf{match-}\mathsf{black}\textbf{-}\mathtt{E}) \quad \frac{}{\varphi; \Gamma \vdash \mathtt{R}(x) \Rightarrow e \downarrow_{\mathsf{black}(i)} C}\;(\textbf{match-}\mathsf{black}\textbf{-}\mathtt{R})$$

Figure 16: The typing rules for matches on red-black trees.

```
fun restore_right (B(R lt, R (rt as (R _,_)))) = R(B lt, B rt)     (* re-color *)
  | restore_right (B(R lt, R (rt as (_,R _)))) = R(B lt, B rt)     (* re-color *)
  | restore_right (B(l, R(R(rll, rlr), rr))) =
          B(R(l, rll), R(rlr, rr))  (* l is black, deep rotate *)
  | restore_right (B(l, R(rl, rr as R _))) =
          B(R(l, rl), rr)           (* l is black, shallow rotate *)
  | restore_right dict = dict
```

Figure 17: restore_right in SML.

```
(lam $arg$.
    case $arg$ of
        E ⇒ E
      | B($l, r$) ⇒
          (case $l$ of
              R($ll, lr$) ⇒
                  case $r$ of
                      R($rl, rr$) ⇒
                          case $rl$ of
                              R _ ⇒ R(B($ll, lr$), B($rl, rr$))                    (* re-color *)
                  ...
            | B($ll, lr$) ⇒
                  case $r$ of
                      R($rl, rr$) ⇒
                          case $rl$ of
                              R($rll, rlr$) ⇒ B(R(B($ll, lr$), $rll$), R($rll, rr$))  (* l is black, deep rotate *)
              ...
      ...) : $\Pi n{:}\mathcal{N}.\,\Pi h{:}\mathcal{N}.\,\mathsf{badRight}(n,h) \to \mathsf{rbt}(n,h)$
```

Figure 18: A fragment of the translation of restore_right into our language.