

Refinement Type Checking

Rowan Davies
August 23, 2003
CMU-CS-??-???

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Software development is a difficult and error prone task. Experience with programming languages with strong static type systems indicates that these type systems catch a large proportion of common errors at compile time. Given this success, it is natural to ask: can we capture and check more of the intended properties of programs?

In this dissertation I describe the development of an extension of the programming language Standard ML which allows many common program invariants to be expressed and checked. This extension is based on refinement types, which combine aspects of standard type systems with mechanisms such as regular grammars, subtyping and intersection types in order to capture some detailed program properties.

I also present a number of extensions to previous work on refinement types which were required to support this development. Firstly, I present a new form of intersection types which are suitable for languages with call-by-value effects, such as Standard ML. Secondly, I present a new approach to checking refinement-type correctness which is based on bi-directional checking, and which avoids previous difficulties with refinement-type inference. Thirdly, I present an extension of refinement types to languages with sequential pattern matching, such as Standard ML.

Chapter 1

Introduction

1.1 Thesis

It is practical to capture and statically check many invariants of computer programs using a new technique called refinement type checking.

1.2 Motivation

Static type systems are a feature of many programming languages. They provide an intuitive mechanism for capturing and checking the fundamental structure of programs. However, programs generally involve many invariants which can not be captured with these type systems. For example, when programming with lists, it is common to have the invariant that a particular result is not an empty list, which generally can not be expressed by the type system.

One approach to capturing such program invariants is to use program analyses, which automatically infer some kinds of invariants, and are often used to support code optimization. Recently automatic program analyses have been applied to dynamically typed languages in order to obtain some of the benefits of static typing, namely catching errors and supporting code maintenance and modularity [FFK⁺96]. Unfortunately, these analyses must make approximations, since the underlying problems are undecidable, and often it is tedious for the programmer to determine whether an apparent error found by the analysis is due to these approximations or due to an actual error in their code. This has the potential to be particularly tedious when a piece of code is modified many times.

[*From here until the end of this section is copied from my proposal, and needs to be changed.*]

The work proposed here aims to demonstrate that another approach to capturing program properties can be used to build practical tools. This approach is to add a more detailed level of types called *refinement types* to a statically-typed language. Each ordinary type may then be refined by many different refinement types, which we also call *sorts* in accordance with the use of this term in order-sorted algebras [DG94]. Refinement types have been studied previously [FP91, Fre94, Pfe93], and allow many more properties of programs to be expressed and checked than conventional type systems. For example, empty and non-empty lists could be defined as refinements of the type list, in which case we also have a sort for functions which map non-empty lists to non-empty lists, which is a refinement of the type of functions from lists to lists. This extends all the advantages of static typing to a wide class of properties of programs, i.e. more errors are caught at compile time, understanding of programs is aided, and there is increased support for modularity, code maintenance, and code optimization.

To allow more than one property to be expressed for a particular part of a program, sorts include an *intersection* operator $\&$, which allows a sort $R \& S$ to be formed from two sorts R and S which refine the same type. For example, we may have a sort for functions which map empty lists to empty lists, and also map non-empty lists to non-empty lists. The operator $\&$ is taken directly from the numerous forms of intersection types (also called conjunctive types) which have been studied in the literature [CDCV81]. In our setting, the presence of intersection allows the construction of a principal sort for each expression, given only its type. Base sorts refining the same base type are naturally ordered by inclusion, which is extended in a standard way to the full type hierarchy. The character of the resulting system is quite different from record or object subtyping.

Previous work on sorts has demonstrated that they would be a useful addition to languages as diverse as the functional programming language ML [FP91, Fre94] and the logical framework LF [Pfe93]. Work on sorts in ML has focused on refinements of datatypes which are defined using recursive definitions, similar to the definitions of the datatypes themselves. Part of that work considered algorithms for sort inference, but this turns out to be problematic because common programs often satisfy many accidental properties which must be reflected in the inferred sort. Further, there is a huge combinatorial explosion in the number of refinements and the size of principal sorts as we move to more complicated (especially higher-order)

types. Experiments with refinement-type systems have thus been limited to small prototypes.

The starting point for this work is the observation that *sort checking* a program which has been annotated with some of the intended sorts should be much easier than sort inference.

1.3 An Introductory Example

To illustrate the practical use of sorts in a language like ML, consider the following example. Suppose we are writing an application which manipulates simple arithmetic expressions. In some parts of this application we maintain the invariant that the expressions manipulated are sums of products, i.e., they do not contain additions as sub-expressions of multiplications. Additionally, in some parts of the application we also maintain the invariant that only ground expressions are manipulated, i.e., expressions which contain no variables. Also, in some parts of the program we maintain both of these invariants.

If we attempt to enforce these invariants using the type system of ML, we will need to create separate types for arbitrary expressions, expressions which are sum of products, ground expressions, and ground expressions which are sum of products. We will then need to duplicate functions which manipulate expressions so that they can be applied to each of the required types, which leads to very awkward code maintenance problems. This is usually too awkward to be feasible, so instead only one type is defined, and the intended invariants remain implicit, and are only documented in comments. To manually check that the invariants actually hold can be a very tedious and error prone process, particularly when the code is modified many times.

The following small code sample shows how these invariants can be captured with sorts:

```
datatype exp = Num of int | Var of string | Plus of exp * exp | Times of exp * exp
datasort prod = Num of int | Var of string | Times of prod * prod
datasort sum_prods = <prod> | Plus of sum_prods * sum_prods
datasort ground = Num of int | Plus of ground * ground | Times of ground * ground

datasort gnd_sp = datasort (ground & sum_prods)

fun sp_times(Plus(st1a, st1b), st2) = Plus(sp_times(st1a, st2), sp_times(st1b, st2))
  | sp_times(st1, Plus(st2a, st2b)) = Plus(sp_times(st1, st2a), sp_times(st1, st2b))
  | sp_times(st1, st2) = Times (st1, st2)
```

```
withsort sp_times :> (sum_prods * sum_prods -> sum_prods) & (gnd_sp * gnd_sp -> gnd_sp)
```

In this example code we define a type `exp` for simple arithmetic expressions. Then we define some refinements of this type using `datasort` declarations, which are part of the proposed extension to ML. These refinements correspond to the restricted forms required by the invariants of our application. The form of `datasort` declarations is similar to `datatype` declarations, except that value constructors may appear many times in the same declaration, and previously declared sorts may be extended, as in the definition of `sum_prods` which extends `prod`. Also, the replicating form of `datasort` declarations allows the body to be an intersection, as illustrated in the definition of `gnd_sp` above.

We then define an example function from our application, `sp_times`, which takes two expressions which are sums of products, and returns a sum of products expression which is equivalent to their product. Additionally, we have the invariant that if both input expressions are also ground, it returns a ground expression. We express these invariants of the `sp_times` function by attaching a `withsort` clause to it, which is another one of our proposed extensions to ML. This clause assigns a sort to the function which is an intersection, which intuitively means that the function must have both of the intersected sorts, each of which corresponds to one of the invariants. The assigned sort will prevent the programmer from applying this function to an argument that is not determined to represent a sum of products (or a ground sum of products), even though the function will not raise an error in this case. This correctly reflects the intention that this function only be applied to sums of products.

Part of the proposed work is to build a sort checker into an existing SML compiler. In fact, an initial implementation of a sort checker has already been built which will successfully check the example code above, thus verifying that the invariants hold. If instead there was an error in our code, such that these invariants did not hold, then sort checking would fail with an informative error message. In fact, because we are only doing sort checking, we can generally give better error messages than those generated when ordinary type *inference* fails, since the latter may propagate incorrect information before detecting the error. However, static debugging based on sort checking is more involved than ordinary static debugging, since `datasort` definitions and sort annotations need to be debugged in addition to the actual code being checked. This debugging process is often beneficial, since it forces the programmer to be aware of the exact invariants of their code, thus increasing confidence that the code is correct.

1.4 Related Work

[*This section is very much out of date.*]

1.4.1 Intersection types

The inclusion of an intersection operator allows sorts to express very precise information about programs by combining many pieces of information into a single sort. Such intersection operators have been extensively studied in the context of typed λ -calculi [CDCV81]. An important property in that context is that in a simply-typed λ -calculus with intersections the well-typed terms are exactly those that have normal forms. This of course means that it is undecidable whether an untyped term has a type.

The use of intersection types in programming languages was first proposed by Reynolds who used them in the language Forsythe [Rey81, Rey96]. In Forsythe intersections are used to represent overloading of arithmetic functions, to represent polymorphism (in the absence of parameteric polymorphism), and to represent records. However, they are generally not used to represent program invariants as is possible with ML sorts, which is not surprising since Forsythe does not have any equivalent of ML datatypes. Forsythe is explicitly typed, and type checking is decidable. The algorithm for Forsythe type checking presented in [Rey96] is somewhat different to that used for ML sort checking in [DP97]. In fact, the Forsythe algorithm relies on the assumption there do not exist base types a, b, c such that $a \& b \leq c$ but not $a \leq c$ nor $b \leq c$, which does not generally hold in the case of sorts.

An important result proved in [Rey96] is that Forsythe type checking is PSPACE-hard. The proof involves translating quantified boolean expressions into Forsythe programs which type check exactly when the boolean expression is typable. Exactly the same technique can be used to show that ML sort checking is PSPACE-hard. In practical terms this means that it is possible for sort checking not to terminate within a reasonable amount of time. However, our experience so far with the current implementation indicates that our sort checking algorithm is likely to be efficient enough for programs encountered in practice.

1.4.2 Regular tree types and set constraints

Part of the original inspiration for adding sorts to ML was the use of regular tree types in logic programming. These types represent subsets of the

Herbrand universe of a logic programming language. Thus, it is very natural for regular tree types to have a subtype ordering corresponding to inclusion of the sets they represent. Regular tree types are defined by regular term grammars, which allow this ordering to be computed relatively efficiently, while also being expressive enough to specify appropriate sets in most cases. This also allows intersections, unions and complements of regular tree types to be computed.

Much of the work on regular tree types, such as [Mis84], has restricted the grammars further so that they are tuple distributive, which means that if $f(A_1, A_2)$ and $f(B_1, B_2)$ are in the set generated by the grammar, then so must $f(A_1, B_2)$ and $f(A_2, B_1)$. This allows the regular tree grammars to be treated as ordinary regular grammars, for which algorithms are well known, but it also reduces the expressiveness considerably. For general regular tree grammars checking inclusion is EXPTIME-hard, which can easily be shown by reduction from the problem of inequivalence of finite tree automata, which was shown to be EXPTIME-complete by Seidl [Sei90]. Despite this, algorithms for comparing and calculating intersections and unions of general regular tree grammars have been proposed by Dart and Zobel [DZ92], and appear to be somewhat practical, though have not been shown to be correct. Aiken and Murphy [AM91] have proposed somewhat different algorithms for a generalization of regular tree grammars which include free variables and negation, though found it necessary to make some approximations for efficiency. An algorithm for the even more general problem of solving set constraints over terms was presented by Aiken and Wimmers [AW92], and shown to have non-deterministic exponential time complexity.

In the context of ML, we would like to be able to define sorts by general regular tree grammars, though we also need to handle parameterized definitions and refinements of function types and reference types in definitions. The algorithm for comparing sorts in the current implementation is complete when sorts are defined using only regular tree grammars, and it uses an apparently more efficient method for handling (non-distributive) tuples than Dart and Zobel. Skalka [Ska97] has formally proved the correctness of an algorithm related to this one, including an extension to parameterized sort definitions. Parameterized sort definitions might also be handled using the free variables allowed in [AM91]. Some hints on how to handle function types in sort definitions can be obtained from an algorithm due to Aiken and Wimmers [AW93] for the more general problem of solving type inclusion constraints, which are essentially set constraints with function types added. Unfortunately their algorithm requires restrictions on occurrences of

intersections and unions, and these restrictions do not seem appropriate for sort definitions.

1.4.3 Soft typing and conditional types

Soft typing aims to bring some of the benefits of statically-typed languages to dynamically typed ones, and was first considered by Reynolds [Rey69], although the term “soft typing” is due to Cartwright and Fagan [CF91]. This is done using analyses or type systems which do not require any type declarations by the programmer, and then automatically inserting dynamic type checks based on the results to guarantee that the resulting program cannot raise a runtime error. Programmers can use the dynamic type checks to help locate possible errors, and some work has been done by Flanagan *et al.* [FFK⁺96] to design a sophisticated interface to help in this process. However, even with such an interface this approach makes it somewhat tedious for a programmer to determine which dynamic type checks are the result of errors in a program, particularly when a large program is modified many times.

Two systems used for soft typing are set-based analysis, due to Heintze [Hei94], and conditional types which are originally due to Reynolds [Rey69] and have been used more recently by Aiken *et al.* [AWL94b]. Both capture quite accurate invariants of programs in some cases, so it is very interesting to compare them to sorts in ML. There do not seem to be major differences in accuracy between these two systems, and so we concentrate on conditional types here to simplify the comparison.

Conditional types allow the control flow information resulting from a conditional (or “case” construct) to be directly represented in a type. Perhaps the most accurate soft-typing systems are those that make use of *constrained types* with conditional types. Constrained types are polymorphic types which include inclusion constraints on their variables, of which many varieties have been studied, starting with Mitchell [Mit84]. These were combined by Aiken *et al.* [AWL94b] to obtain a very expressive type inference system. In fact, this system is expressive enough that at first it appears that it might be able to capture many of the invariants that sorts were designed to express. Alas, the following example illustrates that this is not the case:

```
datatype bitstring = Nil | Zero of bitstring | One of bitstring
datasort evParity = Nil | Zero of evParity | One of odParity
       and odParity = Zero of odParity | One of evParity
```

```

fun append_one Nil = One Nil
  | append_one (Zero bs) = Zero (append_one bs)
  | append_one (One bs) = One (append_one bs)
withsort append_one :> evParity -> odParity
                    & odParity -> evParity

```

This program is well sorted, thus verifying two interesting invariants of the function `append_one`. However, using constrained conditional types for an equivalent function, we obtain the type

$$\begin{aligned}
& \forall \alpha. \alpha \rightarrow \beta \\
& \text{where } \alpha \leq \text{Nil} \cup \text{Zero}(\alpha) \cup \text{One}(\alpha) \\
& \quad \beta = (\text{One}(\text{Nil})?(\alpha \cap \text{Nil})) \\
& \quad \quad \cup (\text{Zero}(\beta)?(\alpha \cap \text{Zero}(1))) \\
& \quad \quad \cup (\text{One}(\beta)?(\alpha \cap \text{One}(1)))
\end{aligned}$$

Roughly, the first constraint means that the argument type α is a particular form of subtype of `bitstring` above, and the second constraint means that if the α has a non-nil intersection with the type containing only `Nil`, then the result type β contains `One(Nil)`, and if α contains any value beginning with `Zero` (1 is an all-inclusive type) then β contains `Zero(x)` for each x in β , and if α contains any value beginning with `One` then β contains `One(x)` for each x in β . This type contains a lot of information, but it certainly does not capture the desired invariants above. In particular, `evParity` and `odParity` do not correspond to valid instantiations of α , since they would violate the first constraint. In fact, the only supertype of either of these types that would be a valid instantiation of α is the one which corresponds to the whole of the type `bitstring`.

A closer analysis of this result indicates that the problem is the lack of polymorphic recursion in the system of [AWL94b]. This suggests that polymorphic recursion would be a very useful addition to this system, though it seems very unlikely that type inference would be decidable for such a system.

1.4.4 Dependent types in ML

Xi and Pfenning [XP97b] have proposed a general schema for adding dependent types to ML, and have applied this schema to the problem of array bounds checking [XP97a]. Their motivations are somewhat similar to those for sorts, namely capturing invariants of programs. However, the invariants that can easily be captured using their form of dependent types are quite

different to those that can be captured using sorts. Also, their schema for type checking ultimately relies on a constraint solver for whatever domain is used to index types, and constraint solving can involve arbitrary theorem proving, which is undecidable in general.

1.5 Introductory examples

[*Here there will be some examples using full SML with datasort declarations. The semantics of sorts will only be explained in a very rough, informal way.*]

Chapter 2

Sort Checking with Standard Intersection Types

This chapter focuses on sort checking with the standard intersection type rules, i.e. without a value restriction and with distributivity. Intersection types with these rules have been extensively studied in the context of λ -calculi (see e.g. [CDCV81]) and in the context of programming languages (e.g. [Rey91][Rey96]). Refinement types with these rules (i.e. with a refinement restriction) have been studied previously in two contexts: an extension of the programming language ML [FP91, Fre94], and an extension of the logical framework LF [Pfe93].

This chapter serves two purposes in the context of the dissertation:

1. To present a bi-directional sort-checking algorithm for this standard form of intersection types, and to compare to previous algorithms for languages with intersection types.
2. For comparison with the intersection types with a value restriction that are used in the remainder of this dissertation.

2.1 Syntax

We follow the Pfenning's presentation of the simply-typed λ -calculus (see Chapter 3 of [Pfe01] also "Logical Frameworks" in "Handbook of Automated Reasoning"¹). We add sorts, separating types and sorts into separate syntactic classes, following Freeman [Fre94], but differing from Pfenning's refinement types for LF [Pfe93]. We call the resulting system $\lambda^{\rightarrow\&}$.

¹*Fix citation.*

[*Comment on the inclusion of \top^A .*]

We include signatures with declarations of sort constants and declarations of subsorting relationships, following Pfenning [Pfe93], but differing from Freeman [Fre94]. We allow subsorting declarations of the form $R \leq_a S$ where R and S are refinements of a type constant a . This extends the declarations in [Pfe93] which have the form $r \leq s$. The extension is necessary so that all finite lattices of base sorts can be declared (see Section 2.5).

We use a, b for type constants, r, s for sort constants, c for object constants and x for object variables.

Types	A	$::=$	$a \mid A_1 \rightarrow A_2$
Type Contexts	Γ	$::=$	$\cdot \mid \Gamma, x:A$
Objects	M	$::=$	$c \mid x \mid \lambda x:A. M \mid M_1 M_2$
Sorts	R	$::=$	$r \mid R_1 \rightarrow R_2 \mid R_1 \& R_2 \mid \top_A$
Sort Contexts	Δ	$::=$	$\cdot \mid \Delta, x \in R$
Declarations	D	$::=$	$a:\text{type} \mid c:A \mid r \sqsubset a \mid c \in R \mid R \leq_a S$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, D$

We use A, B for types, R, S for sorts and M, N for objects. We write $\{N/x\}M$ for the result of substituting N for x in M , renaming bound variables as necessary to avoid the capture of free variables in N . We use the notation $\mathcal{D} :: J$ to indicate that \mathcal{D} is a derivation of judgment J .

We require variables to appear at most once in a type or sort context. Similarly, we require signatures to include at most one declaration $a:\text{type}$ for each a , at most one declaration $r \sqsubset a$ for each r , and at most one declaration $c:A$ and one declaration $c \in R$ for each c .

There are a number of reasonable alternatives to the above formulation of signatures.

- We could have separate signatures for type and sort level declarations.
- We could remove declarations of the form $c:A$ and instead make $c:A$ a consequence of $c \in R$ when $R \sqsubset A$.
- We could allow $c \in R_1$ and $c \in R_2$ to appear in the same signature (which would be equivalent to $c \in R_1 \& R_2$) as is done in Pfenning's refinements for LF [Pfe93].

These alternatives seem to be mostly cosmetic.

2.2 Validity Judgments

The validity judgments for $\lambda^{\rightarrow\&}$ extend those of [Pfe01]. The validity judgment for signatures is extended to the new forms of declarations $r \sqsubset a$, $c \in R$ and $R \leq_a S$. This judgment depends upon the standard validity judgment for types with respect to a valid signature. We have a new judgment for the validity of sorts as refinements of a particular valid type with respect to a valid signature.

$\vdash \Sigma \text{ Sig}$ Σ is a valid signature
 $\vdash_{\Sigma} A : \text{type}$ A is a valid type
 $\vdash_{\Sigma} R \sqsubset A$ R is a valid refinement of type A .

Valid Signatures

$$\begin{array}{c}
 \frac{}{\vdash \cdot} \text{sigemp} \qquad \frac{\vdash \Sigma \text{ Sig}}{\vdash \Sigma, a:\text{type} \text{ Sig}} \text{sigtyp} \qquad \frac{\vdash \Sigma \text{ Sig} \quad \vdash_{\Sigma} A : \text{type}}{\vdash \Sigma, c:A \text{ Sig}} \text{sigobj} \\
 \\
 \frac{\vdash \Sigma \text{ Sig} \quad \vdash_{\Sigma} a : \text{type}}{\vdash \Sigma, (r \sqsubset a) \text{ Sig}} \text{sigsrt} \qquad \frac{\vdash \Sigma \text{ Sig} \quad \vdash_{\Sigma} c : A \quad \vdash_{\Sigma} R \sqsubset A}{\vdash \Sigma, c \in R \text{ Sig}} \text{sigobjsr} \\
 \\
 \frac{\vdash \Sigma \text{ Sig} \quad \vdash_{\Sigma} R \sqsubset a \quad \vdash_{\Sigma} S \sqsubset a}{\vdash \Sigma, (R \leq_a S) \text{ Sig}} \text{sigsub}
 \end{array}$$

Valid Types

$$\frac{a:\text{type in } \Sigma}{\vdash_{\Sigma} a : \text{type}} \text{typcon} \qquad \frac{\vdash_{\Sigma} A : \text{type}}{\vdash_{\Sigma} B : \text{type}} \text{arrow}$$

Valid Refinements

$$\frac{\vdash_{\Sigma} a : \text{type}}{\vdash_{\Sigma} r \sqsubset a} \text{srtcon} \qquad \frac{\vdash_{\Sigma} R \sqsubset A \quad \vdash_{\Sigma} S \sqsubset B}{\vdash_{\Sigma} R \rightarrow S \sqsubset A \rightarrow B} \text{srtarrow}$$

$$\frac{\vdash_{\Sigma} R_1 \sqsubset A \quad \vdash_{\Sigma} R_2 \sqsubset A}{\vdash_{\Sigma} R_1 \& R_2 \sqsubset A} \text{srtinter}$$

[Add \top^A here.]

A direct consequence of these definitions is that every sort refines at most one type.

We say that a sort R is *well-formed* if it refines some type A . In what follows, we will only be interested in well-formed sorts, and so when we use the term “sort” we implicitly mean “well-formed sort”. We say that two sorts are *compatible* if they refine the same type.²

The judgments for validity of type contexts and objects are completely standard. We add a judgment for validity of sort contexts refining a valid type context. Each of these judgments is with respect to a valid signature Σ .

$$\begin{array}{l} \vdash_{\Sigma} \Gamma \text{ Ctx} \quad \Gamma \text{ is a valid context} \\ \Gamma \vdash_{\Sigma} M : A \quad M \text{ is a valid object of type } A \text{ in valid type context } \Gamma \\ \vdash_{\Sigma} \Delta \sqsubset \Gamma \quad \Delta \text{ is a valid refinement of valid type context } \Gamma. \end{array}$$

Valid Type Contexts

$$\frac{}{\vdash_{\Sigma} \cdot \text{Ctx}} \text{ctxemp} \qquad \frac{\vdash_{\Sigma} \Gamma \text{ Ctx} \quad \vdash_{\Sigma} A : \text{type}}{\vdash_{\Sigma} \Gamma, x:A \text{ Ctx}} \text{ctxobj}$$

²I might adapt some more of the discussion on the refinement relation in the next chapter.

Valid Objects

$$\begin{array}{c}
\frac{c:a \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c : a} \text{objcon} \qquad \frac{x:A \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x : A} \text{objvar} \\
\\
\frac{\Gamma, x:A \vdash_{\Sigma} M : B}{\Gamma \vdash_{\Sigma} \lambda x:A. M : A \rightarrow B} \text{objlam} \qquad \frac{\Gamma \vdash_{\Sigma} M : A \rightarrow B \quad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} M N : B} \text{objapp}
\end{array}$$

Valid Sort Contexts

$$\frac{}{\vdash_{\Sigma} \cdot \sqsubset \cdot} \text{sctxemp} \qquad \frac{\vdash_{\Sigma} \Delta \sqsubset \Gamma \qquad \vdash_{\Sigma} R \sqsubset A}{\vdash_{\Sigma} (\Delta, x \in R) \sqsubset (\Gamma, x:A)} \text{sctxobj}$$

We do not include the sorting judgment for objects here because it is not a validity judgment. Instead it is a sort *assignment* judgment that assigns sorts to terms that are already judged to be valid using the typing judgment.

2.3 Reduction and Canonical Forms

[*There's not much to say here, since the level of type already guarantees canonical forms. I might say something about the canonical forms with certain sorts, particularly base sorts.*]

2.4 Declarative Subsorting

The subsorting judgment has the following form, where R and S must be compatible sorts.

$$\vdash_{\Sigma} R \leq S \quad \text{Sort } R \text{ is a subsort of } S$$

The subsorting rules are standard (See e.g. [Rey91]). The signature Σ is fixed throughout these rules, and we omit the \vdash_{Σ} for brevity here and as appropriate in the remainder of this chapter.

$$\begin{array}{c}
\frac{R \leq_a S \text{ in } \Sigma}{R \leq S} \text{sub_base} \\
\\
\frac{}{R \leq R} \text{sub_reflex} \quad \frac{R_1 \leq R_2 \quad R_2 \leq R_3}{R_1 \leq R_3} \text{sub_trans} \\
\\
\frac{}{R \ \& \ S \leq R} \text{sub_conjL1} \quad \frac{}{R \ \& \ S \leq S} \text{sub_conjL2} \\
\\
\frac{R \leq S \quad R \leq S'}{R \leq S \ \& \ S'} \text{sub_conjR} \\
\\
\frac{R' \leq R \quad S \leq S'}{R \rightarrow S \leq R' \rightarrow S'} \text{sub_arrow} \\
\\
\frac{}{(R \rightarrow S) \ \& \ (R \rightarrow S') \leq R \rightarrow (S \ \& \ S')} \text{sub_dist}
\end{array}$$

[Add \top^A here.]

2.5 Sort Equivalence and Finiteness

If $R \leq S$ and $S \leq R$ then we say R and S are *equivalent* sorts. We use the notation $R \cong S$ for this relation, which is an equivalence relation: it is reflexive by rule `sub.reflex` (in each direction) and transitive by rule `sub.trans` (in each direction). Our main motivation for introducing this definition is that there are only a finite number of refinements of each type modulo sort equivalence, which can be proved by induction on the structure of the type refined by R and S . We omit a formal proof here. Section 3.7 includes a formal proof of a corresponding finiteness theorem in the absence of the rule `sub.dist`. Since the removal of this rule can only result in more distinct refinements, this proof can easily be adapted to the situation in this chapter. See [Fre94] for a similar finiteness theorem for a similar system of sorts.

2.6 Relating Base Refinements and Finite Lattices

For a particular signature Σ , the finiteness of the distinct refinements of a base type a implies that they form a lattice. Refinements r_1, r_2 have a greatest lower bound $r_1 \& r_2$ and their least upper bound is the intersection of a finite set containing all distinct upper bounds (up to sort equivalence), i.e.

$$\&\{s : r_1 \leq s \text{ and } r_2 \leq s\}.$$

Further, for any finite lattice with elements l_1, \dots, l_n we can construct a signature Σ that includes a base type a with distinct refinements matching the structure of l_1, \dots, l_n . We do this by including the following declarations in Σ :

$$\left. \begin{array}{l} r_i \sqsubset a \\ r_i \& r_j \leq^a r_k \\ r_k \leq^a r_i \& r_j \end{array} \right\} \begin{array}{l} \text{for each lattice element } l_i \text{ (choosing distinct } r_i) \\ \text{for each } l_i, l_j \text{ (} i = 1, \dots, n \text{ and } j = 1, \dots, n) \\ \text{with greatest lower bound } l_k. \end{array}$$

Thus, the base refinement structures that can be expressed with our signatures correspond exactly to the set of finite lattices. This matches the structures allowed by Freeman [Fre94], who directly assumes that the base refinements have a partial order and an intersection operator that satisfy the properties of a lattice. However, our assumptions are isolated in a signature, following Pfenning [Pfe93], which seems more elegant than the explicit global assumptions in Freeman's presentation. Our signatures extend those of Pfenning by including declarations of the form $R \leq^a S$ instead of the form $r \leq s$. This extension is necessary to allow all finite lattices to be declared. E.g. the following signature has no counterpart in the system presented by Pfenning:

$$\Sigma = a:\text{type}, c:a, r_1 \sqsubset a, r_2 \sqsubset a, r_3 \sqsubset a, (r_2 \& r_3 \leq_a r_1).$$

3

2.7 Declarative Sorting

The sort assignment judgment has the following form, where $\Gamma \vdash M : A$, $\Delta \sqsubset \Gamma$ and $R \sqsubset A$.⁴

³Maybe present this more formally?

⁴I prefer to make these part of the form of the judgment rather than proving $\Delta \vdash M \in R$ implies $\Gamma \vdash M : A$.

$\Delta \vdash M \in R$ Term M has sort R in context Δ .

The sorting rules are very similar to those for a system with general intersection types (e.g., see [Rey91]). Here each abstraction includes the type of the variable, so the choice of the sort for the variable is restricted to refinements of this type.

$$\frac{x \in R \text{ in } \Delta}{\Delta \vdash x \in R} \text{srt_var} \quad \frac{\Delta, x \in R \vdash M \in S \quad R \sqsubset A}{\Delta \vdash \lambda x:A. M \in R \rightarrow S} \text{srt_lam}$$

$$\frac{\Delta \vdash M \in R \rightarrow S \quad \Delta \vdash N \in R}{\Delta \vdash MN \in S} \text{srt_app}$$

$$\frac{\Delta \vdash M \in R \quad \Delta \vdash M \in S}{\Delta \vdash M \in R \ \& \ S} \text{srt_conj} \quad \frac{\Delta \vdash M \in R \quad R \leq S}{\Delta \vdash M \in S} \text{srt_subs}$$

Our calculus satisfies the usual substitution lemmas with respect to sorts, subject reduction and sort preservation theorems with respect to β -reduction. Details are omitted here for brevity, since they have appeared elsewhere.⁵ Proofs of similar results for a system with a value restriction appear in the next chapter.

We recall the following theorem and its simple proof.

Theorem 2.7.1 (Principal sorts) *If $\Delta \vdash M \in R$ then there exists S such that $\Delta \vdash M \in S$ and for all R' such that $\Delta \vdash M \in R'$ we have $S \leq R'$.*

Proof: Choose S to be the intersection of all sorts S' (up to equivalence) such that $\Delta \vdash M \in S'$. Clearly S is unique up to sort equivalence, and we call S the *principal* sort of M with respect to Δ . \square

2.8 Algorithmic Subsorting

The declarative sorting and subsorting rules are quite intuitive, but they do not specify a strategy for checking a term. In this section we present an algorithm for determining subsorting. We address the more difficult issue of algorithmic sort checking later.

⁵Check this, and add citations here.

Our algorithm is similar to that used by Freeman [Fre94]. Other algorithms for determining subsorting or subtyping in the presence of intersection types include that designed by Reynolds [Rey91] for the programming language Forsythe.⁶ Interestingly, this algorithm cannot be easily extended to our base subsorting relations.⁷ For example, if we have the signature from Section 2.5:

$$\Sigma = a:\text{type}, c:a, r_1 \sqsubset a, r_2 \sqsubset a, r_3 \sqsubset a, (r_2 \& r_3 \leq_a r_1)$$

then we can form the following subsorting derivation:

$$\frac{\frac{}{(r_1 \rightarrow r_2) \& (r_1 \rightarrow r_3) \leq r_1 \rightarrow (r_2 \& r_3)} \quad \frac{r_2 \& r_3 \leq r_1 \text{ in } \Sigma}{r_2 \& r_3 \leq r_1}}{(r_1 \rightarrow r_2) \& (r_1 \rightarrow r_3) \leq r_1 \rightarrow r_1.}$$

However, an algorithm based on the Forsythe approach would incorrectly determine that this subsorting instance is false. Roughly, this is because the algorithm depends upon the occurrence of an intersection sort within the sort on the right to determine potential uses of the distributivity rule. But when we have a declaration like $r_2 \& r_3 \leq r_1$ there are some subsorting instances which require the distributivity rule but have no such occurrence.

One possible solution to this is to replace the signature with

$$\Sigma' = a:\text{type}, c:a, r_2 \sqsubset a, r_3 \sqsubset a, s_2 \sqsubset a, s_3 \sqsubset a, r_2 \leq_a s_2, r_3 \leq_a s_3$$

and then replace all occurrences of r_1 with $s_2 \& s_3$. This correctly maintains the relationships between r_1 , r_2 , r_3 and their intersections. However, a transformation like this would result in an exponential increase in the size of the base sort lattice in some cases (e.g. when the lattice is flat).⁸

The subsorting algorithm that we present here is a variant of that presented by Freeman [Fre94]. More precisely, it is the “more efficient” variant mentioned on page 119 of [Fre94]. Freeman focuses on a less efficient algorithm because it is easier to prove correct, and fits better with the representations of sorts used in his sort inference algorithm. However, the less efficient algorithm seems less likely to scale to complex and higher-order types because it requires explicit enumeration of the refinements of a type.

⁶*I need to check whether it's appropriate to add here [CD78], [AWL94a], [TDMW97], ...*

⁷An algorithm based on a similar approach is described more formally by Pierce [Pie91, Pie93]

⁸*Elaborate? Show the case of a flat lattice?*

Our sub-sorting algorithm is described by a judgment with algorithmic rules, and depends on an ancillary judgment which synthesizes the result sort when a function sort is projected on a particular argument sort. We also make use of the negation of this judgment.⁹ The judgments have the following forms:

[*NEXT: Add algorithmic subsorting of base sorts to the below. This is a little messy with this presentation. One reasonable way is to consider them as generated from finite sets of base sorts ρ (denoting $\& \rho$), and “complete” these sets by adding ρ_2 to ρ whenever there is a declaration $\rho_1 \leq_a \rho_2$ in Σ with $\rho_1 \subseteq \rho$.]*

$R \trianglelefteq S$ Sort R is determined to be a sub-sort of S ,
 where R and S are compatible.

$R \trianglelefteq S_1 \gg S_2$ Function sort R when projected on argument sort S_1
 yields result sort S_2 , where $R \sqsubset A_1 \rightarrow A_2$, $S_1 \sqsubset A_1$,
 and $S_2 \sqsubset A_2$.

$$\frac{R \trianglelefteq s^a}{R \trianglelefteq s^a} \text{subalg_base} \qquad \frac{R \trianglelefteq S \quad R \trianglelefteq S'}{R \trianglelefteq S \ \& \ S'} \text{subalg_conj}$$

$$\frac{R @ S_1 \gg R_2 \quad R_2 \trianglelefteq S_2}{R \trianglelefteq S_1 \rightarrow S_2} \text{subalg_arrow}$$

$$\frac{R' \trianglelefteq R}{R \rightarrow S @ R' \gg S} \text{apsrt_arrow} \qquad \frac{R @ S_1 \gg S_2 \quad R' @ S_1 \gg S'_2}{R \ \& \ R' @ S_1 \gg S_2 \ \& \ S'_2} \text{apsrt_cnj}$$

$$\frac{R @ S_1 \gg S_2 \quad R' @ S_1}{R \ \& \ R' @ S_1 \gg S_2} \text{apsrt_cnj1} \qquad \frac{R @ S_1 \quad R' @ S_1 \gg S'_2}{R \ \& \ R' @ S_1 \gg S'_2} \text{apsrt_cnj2}$$

[*Everything that follows in this chapter is copied from elsewhere (mostly my thesis proposal and the POPL'98 submission)]*

To show that the algorithmic sub-sorting judgment is equivalent to the original one, we need the following lemmas:

Lemma 2.8.1 (Conjuncts) *if S is conjunct of R then*

⁹Maybe use top instead here, so we only need the negation of \trianglelefteq judgment. Otherwise, explain this further, and maybe cite Alberto's work.

1. $R \trianglelefteq S$.
2. if $S = S_1 \rightarrow S_2$ then $R @ S_1 \gg R_2$ and S_2 is a conjunct of R_2 .

Proof: By induction on S and R , making use of the algorithmic sub-sorting lemma and consistency of base sorting. \square

Corollary 2.8.2 (Reflexivity) *For every well-formed sort S we have $S \trianglelefteq S$.*

Lemma 2.8.3 (Conjunction) *if $R \trianglelefteq S$ then $R \& R' \trianglelefteq S$ and $R' \& R \trianglelefteq S$.*

Proof: By induction on the derivation of $R \trianglelefteq S$, making use of the consistency of base sorting. \square

Lemma 2.8.4 (Transitivity)

1. if $R_1 \trianglelefteq R_2$ and $R_2 \trianglelefteq R_3$ then $R_1 \trianglelefteq R_3$
2. if $R_1 \trianglelefteq R_2$ and $R_2 @ R_3 \gg S_2$ then $R_1 @ R_3 \gg S_1$ and $S_1 \trianglelefteq S_2$.
3. if $R_1 @ R_2 \gg S_2$ and $R_3 \trianglelefteq R_2$ then $R_1 @ R_3 \gg S_3$ and $S_3 \trianglelefteq S_2$.

Proof: By induction on the sum of the lengths of R_1 , R_2 and R_3 , making use of the above lemmas. \square

Theorem 2.8.5 (Algorithmic sub-sorting correctness) *$R \trianglelefteq S$ if and only if $R \leq S$.*

Proof: From left to right is proved by straight-forward induction on derivations. From right to left is also proved by induction on derivations, making use of the above lemmas. \square

2.9 Bidirectional Sort Checking

10 11

We now give our algorithm for sort checking annotated terms, presented as a judgment with algorithmic rules, and with two ancillary judgments,

¹⁰Add a comment on the relationship between bidirectional checking and canonical forms.

¹¹So far this is just a copy of what was in my thesis proposal.

one of which synthesizes the principal sort of a term of the form I , and one of which projects a function sort on an argument term. We also need the negation of the projection judgment. The idea of these judgments is that the sort of each variable in the context is always known, and to check whether a term has an intersection sort we simply check that it has each of the conjuncts. There is some small flexibility here as to exactly when intersections are broken down, and in practice we break them down as late as possible. We have omitted the typing rules for annotated terms, though they are completely standard, with the additional requirements that the sort annotations are required to refine the corresponding types.

$\Delta \vdash C \llcorner R$	Term C is determined to have sort R in context Δ , where $\Delta \sqsubset \Gamma$, $R \sqsubset A$, $\Gamma \vdash C : A$.
$\Delta \vdash I \gg R$	Term I has principal sort R in context Δ , where $\Delta \sqsubset \Gamma$, $R \sqsubset A$ and $\Gamma \vdash C : A$.
$\Delta \vdash R @ C \gg S$	Sort R when projected on term C yields result sort S , where $\Delta \sqsubset \Gamma$, $R \sqsubset A_1 \rightarrow A_2$, $\Gamma \vdash C : A_1$ and $S \sqsubset A_2$.
$\Delta \vdash R \not@ C$	Sort R can not be projected on term C , where $\Delta \sqsubset \Gamma$, $R \sqsubset A_1 \rightarrow A_2$ and $\Gamma \vdash C : A_1$.

$$\begin{array}{c}
\frac{\Delta \vdash C \llcorner R \quad \Delta \vdash C \llcorner S}{\Delta \vdash C \llcorner R \ \& \ S} \text{dn_conj} \qquad \frac{\Delta, x \in R \vdash C \llcorner S \quad R \sqsubset A}{\Delta \vdash \lambda x:A. C \llcorner R \rightarrow S} \text{dn_lam} \\
\frac{\Delta \vdash I \gg R \quad \Delta, x \in R \vdash C \llcorner S}{\Delta \vdash \text{let } x = I \text{ in } C \llcorner S} \text{dn_letA} \qquad \frac{\Delta \vdash C_1 \llcorner R \quad \Delta, x \in R \vdash C_2 \llcorner S}{\Delta \vdash \text{let } x \in R = C_1 \text{ in } C_2 \llcorner S} \text{dn_letR} \\
\frac{\Delta \vdash I \gg R \quad R \trianglelefteq S}{\Delta \vdash I \llcorner S} \text{dn_atom}
\end{array}$$

.....

$$\frac{x \in R \text{ in } \Delta}{\Delta \vdash x \gg R} \text{up_var} \qquad \frac{\Delta \vdash I \gg R \quad \Delta \vdash R @ C \gg S}{\Delta \vdash IC \gg S} \text{up_app} \qquad \frac{\Delta, x \in R \vdash C \llcorner R}{\Delta \vdash \text{fix } x \in R. C \gg R} \text{up_fix}$$

$$\begin{array}{c}
\dots\dots\dots \\
\frac{\Delta \vdash C \lll R_1}{\Delta \vdash R_1 \rightarrow R_2 @ C \gg R_2} \text{aptm_arrow} \quad \frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' @ C \gg S'_2}{\Delta \vdash R \& R' @ C \gg S_2 \& S'_2} \text{aptm_conj} \\
\frac{\Delta \vdash R @ C \gg S_2 \quad \Delta \vdash R' @ C}{\Delta \vdash R \& R' @ C \gg S_2} \text{aptm_conj1} \quad \frac{\Delta \vdash R @ C \quad \Delta \vdash R' @ C \gg S'_2}{\Delta \vdash R \& R' @ C \gg S'_2} \text{aptm_conj2}
\end{array}$$

The following theorem shows that our algorithm is sound, by relating sort checking for an annotated term with the sorts of the term obtained by erasing the sort annotations (and replacing them with types for occurrences of **fix**). This erasure process defines a function $erase(\cdot)$, which is a left inverse of the function defined by the annotation process (excluding expansion).

Theorem 2.9.1 (Algorithmic sort-checking soundness)

1. if $\Delta \vdash C \lll R$ then $\Delta \vdash erase(C) \in R$.
2. if $\Delta \vdash I \ggg R$ then $\Delta \vdash erase(I) \in R$.
3. if $\Delta \vdash R @ C \gg S_2$ then there exists S_1 such that $\Delta \vdash erase(C) \in S_1$ and $R \leq S_1 \rightarrow S_2$.

Proof: By induction on the corresponding derivations, making use of inversion on the algorithmic form of subsorting. \square

Theorem 2.9.2 (Algorithmic sort-checking completeness)

1. if $\mathcal{D} :: \Delta \vdash M \in R$ and $ann(M, \mathcal{D}) = C$ then $\Delta \vdash C \lll R$.
2. if $\mathcal{D} :: \Delta \vdash M \in S$ and $ann(M, \mathcal{D}) = I$ then there exists R such that $\Delta \vdash I \ggg R$ and $R \trianglelefteq S$.
3. if $\mathcal{D} :: \Delta \vdash M \in S_1$ and $R \leq S_1 \rightarrow S_2$ and $ann(M, \mathcal{D}) = C$ then there exists S'_2 such that $\Delta \vdash R @ C \gg S'_2$ and $S'_2 \trianglelefteq S_2$.

Proof: By induction on the corresponding derivations, making use of the appropriate replacement lemma, and a lemma validating subsumption in the algorithmic system. \square

We have also proved a slightly stronger completeness theorem, stating that our sort-checking algorithm is complete with respect to the appropriate declarative sorting rules for annotated terms.

2.10 Annotation

[This should follow the corresponding section in the next chapter closely.]

Chapter 3

Sort Checking with a Value Restriction

Our first major challenge in extending sorts to Standard ML is that standard formulations of intersection type systems are unsound in the presence of call-by-value computational effects. The following example demonstrates the unsoundness that results when we naively extend sorts to the reference types of ML:

Example 3.0.1

```
datatype bool = true | false
datasort tt = true and ff = false

(*[ val r :> (bool ref) & (tt ref) ]*)
val r = ref true

(*[ val x :> tt ]*)
val x = (r := false ; !r)
```

Here we define two refinements of the boolean type `bool`, and then create a reference cell `r` containing `true`. Now, `true` has sort `tt` so `r` has sort `tt ref`. Similarly, `true` has sort `bool` so `r` can also be assigned the sort `bool ref`. The standard rule for intersection introduction then allows us to assign the intersection of these sorts to `r`. But this is unsound, because the second part of this sort allows us to update `r` with `false`, while the first part requires that reading from the cell always returns `true`.

This chapter presents $\lambda^{v \rightarrow \&}$, a ¹ λ -calculus with sorts which is suitable for extension with effects.² This is achieved via a *value-restriction* on the introduction of intersection polymorphism. Our restriction is similar to the value restriction on parametric polymorphism proposed by Wright [Wri95] and included in the revised definition of Standard ML [MTHM97] to prevent unsound uses of parametric polymorphism.

It is not immediately clear that we can add a restriction on intersections and still have a sensible system: some basic properties might fail to hold, such as preservation of sorts under reduction. The situation is more complicated than the case of parametric polymorphism in ML because we have a subsorting relationship between sorts.³ The following example shows that introducing a value restriction on intersections is not enough to guarantee soundness in the presence of effects:

Example 3.0.2

```
(*[ val f :> (unit -> (bool ref)) & (unit -> (tt ref)) ]*)
fun f () = ref true

(*[ val r :> (bool ref) & (tt ref) ]*)
val r = f ()
```

Here the sort for `r` is obtained by using subsumption on the sort for `f` with an instance of the standard distributivity subtyping rule for intersection types:

$$(R \rightarrow S_1) \& (R \rightarrow S_2) \leq R \rightarrow (S_1 \& S_2)$$

This example is unsound for similar reasons to the previous example.

We can analyze this situation by considering effectful functions as equivalent to pure functions that return an “effectful computation” $R \Rightarrow \star S$, as in the computational meta-language proposed by Moggi [Mog89, Mog91]. Then, the following subsorting is sound (it is a substitution instance of the standard distributivity rule for pure functions).

$$(R \Rightarrow \star S_1) \& (R \Rightarrow \star S_2) \leq R \Rightarrow (\star S_1) \& (\star S_2)$$

However, the unsound distributivity rule for effectful functions corresponds to the following.

$$(R \Rightarrow \star S_1) \& (R \Rightarrow \star S_2) \leq R \Rightarrow \star (S_1 \& S_2)$$

¹ *call-by-value?*

² *Perhaps use $\&$ instead of \wedge to indicate the presence of the refinement restriction?*

³ *Check reference on parametric polymorphism with subtyping and no distributivity.*

This is unsound because in general $(\star S_1) \& (\star S_2)$ is not a subsort of $\star(S_1 \& S_2)$. For example, if we make effectful computations explicit using \star then the computation

`ref true`

in the examples above could be assigned the sort

`\star(bool ref) & \star(tt ref)`

but not

`\star((bool ref) & (tt ref)).`

Roughly this is because the computation creates a reference cell when it executes, and we can designate that the reference cell always contains a value with sort `bool`, and similarly we can designate that the cell always contains a value with sort `tt`, but we can not designate both at the same time when the cell is created. This explanation applies both to the unsoundness of the distributivity rule and to the unsoundness of intersections without a value restriction.

The other standard subtyping rules for intersection and function types are sound with call-by-value effects. Roughly, this is because none of them involve both functions and intersections, and so none of them require

$$(\star S_1) \& (\star S_2) \leq \star(S_1 \& S_2)$$

to be sound when we add effectful functions. We thus discard this distributivity rule, leaving a system with a pleasing orthogonality between the subsorting rules: each rule only involves a single sort construct.⁴

The main purpose of this chapter is to consider sorts with a value restriction and no distributivity in a simple and general context. We thus focus on a λ -calculus with arbitrary lattices of refinements for each base type. We prove some basic results, such as finiteness of refinements up to equivalence. We also give a relatively simple algorithm for determining subsorting and prove it correct. We then demonstrate that our calculus and sorting rules are at least basically well-behaved by proving a subject reduction theorem.

In Chapter 4 we formally demonstrate the soundness of the extension of $\lambda^{v \rightarrow \&}$ to a call-by-value language with a standard feature involving effects, namely mutable references. It is our intention that $\lambda^{v \rightarrow \&}$ could also

⁴*I'm now not so sure that this orthogonality is enough reason to discard the distributivity rules for e.g. products and sums. Instead, I think the lack of orthogonality is an artifact of the form of the subtyping judgment. E.g. in the work of Denney all the rules are orthogonal, but the equivalents of the distributivity rules can be proven as theorems.*

serve as the basis for other languages with intersection types but without the distributivity subtyping rule. For example a sort system for LF is currently being investigated⁵ which avoids distributivity so that the subsorting relation is simpler and robustly extends to a variety of function spaces.

A major obstacle encountered in previous work on refinements for ML was the lack of a practical algorithm for inferring principal refinements (see Freeman [Fre94]). This appears to be due to a huge combinatorial explosion in the number of refinements as we move to more complicated types. The new features of $\lambda^{v \rightarrow \&}$ do not result in any obvious simplification of these problems, and in fact the absence of distributivity leads to an increase in the number of distinct refinements. At the end of this chapter we make use of the simple and general context of $\lambda^{v \rightarrow \&}$ to present the core of the most important algorithm in this dissertation, namely bi-directional sort checking. This algorithm has been used as the basis of an implementation of a practical sort checker for Standard ML, which will be described further in Chapter 7.

3.1 Syntax

We now present the syntax for $\lambda^{v \rightarrow \&}$. We separate types and sorts into separate syntactic classes, following Freeman [Fre94]. We assume that there are some base types, denoted by a , each of which is refined by a finite number of base sorts, denoted by r^a . We use a signature Σ to capture typing and sorting assumptions for constants.

We use \rightarrow in this chapter for functions, while in later chapters we use \multimap to indicate effectful functions. This is because we expect the results in this chapter to have relevance for other function spaces, even though we are mostly interested in effectful functions.

Types	A	$::=$	$a \mid A_1 \rightarrow A_2$
Type Contexts	Γ	$::=$	$\cdot \mid \Gamma, x:A$
Terms	M	$::=$	$c \mid x \mid \lambda x:A. M \mid M_1 M_2$
Sorts	R	$::=$	$r^a \mid R_1 \rightarrow R_2 \mid R_1 \& R_2 \mid \top^A$
Sort Contexts	Δ	$::=$	$\cdot \mid \Delta, x \in R$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, c:A \mid \Sigma, c \in R$

We use A, B for types, R, S for sorts and M, N for terms. We use a, b for base types, r^a, s^b for base sorts and c for constants. We write $[N/x]M$

⁵Reference? Personal communication?

for the result of substituting N for the variable x in M , renaming bound variables as necessary to avoid the capture of free variables in N . We use the notation $\mathcal{D} :: J$ to indicate that \mathcal{D} is a derivation of judgment J .

3.2 Typing

The typing judgment for $\lambda^{v \rightarrow \&}$ has the following (standard) definition:

$\Gamma \vdash_{\Sigma} M : A$ Term M has type A in context Γ with signature Σ .

$$\frac{c:a \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c : a} \text{tp_const} \qquad \frac{x:A \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x : A} \text{tp_var}$$

$$\frac{\Gamma, x:A \vdash_{\Sigma} M : B}{\Gamma \vdash_{\Sigma} \lambda x:A. M : A \rightarrow B} \text{tp_lam} \qquad \frac{\Gamma \vdash_{\Sigma} M : A \rightarrow B \quad \Gamma \vdash_{\Sigma} N : A}{\Gamma \vdash_{\Sigma} M N : B} \text{tp_app}$$

3.3 Reduction

Reduction for $\lambda^{v \rightarrow \&}$ is defined using the standard β -value rule (following Plotkin [Plo75]).

First, we need to distinguish some terms as *values*:

$$\begin{aligned} \text{Values } V & ::= c \mid x \mid \lambda x:A. M \\ \text{Atoms } V & ::= c \mid x \mid \lambda x:A. M \end{aligned}$$

6 7

There are a number of other sensible definitions of values. The essential feature in the development that follows is that there is a distinguished class of terms to which we restrict the intersection introduction rule. We will point out interesting variations during this development, in particular in Section 3.9.⁸

⁶Maybe we should allow applications of constants here? Or perhaps applying a constant (like “ref”) could have effects? I’m tending towards allowing $cV_1 \dots V_n$ as a value. I still need to check what changes this will require in the rest of this chapter, but I think they should be minor. Distributivity should hold for constants: maybe the sorts/types of constants should be written with a “pure function” arrow?

⁷OLD EDNOTE:I’m now not so sure that it’s worth including constants here, since they act just like variables unless we have a “case” construct or allow applications of constants as values (I guess this means allowing constants to have types involving pure functions).

⁸Actually, so far there is only one place where variations are mentioned.

In the context of effectful functions, values are those terms for which the absence of effects is syntactically immediate.

We have the following standard β -value reduction rule. It allows an argument to be substituted into the body of a function, provided that the argument is a value: ⁹ ¹⁰

$$\frac{}{(\lambda x:A. M) V \mapsto [V/x]M} \beta_v$$

We also have the compositional rules:

$$\frac{M \mapsto N}{\lambda x:A. M \mapsto \lambda x:A. N} \text{reduce_lam}$$

$$\frac{M \mapsto M'}{M N \mapsto M' N} \text{reduce_app1} \quad \frac{N \mapsto N'}{M N \mapsto M N'} \text{reduce_app2}$$

So far $\lambda^{v \rightarrow \&}$ is a standard call-by-value λ -calculus, and we omit the proofs of standard results such as subject reduction with respect to types. Our main interest is in the refinements of the standard types.

3.4 Base Sort Lattices

To smoothly account for intersections of base sorts, we assume that we have a lattice of base sorts refining each base type. Formally, we assume there is a binary operator $\overset{\text{def}^a}{\&}$ for each base type a that maps each pair of base refinements r_1^a, r_2^a to a base sort r_3^a . We also assume that there is a distinguished base sort $\overset{\text{def}^a}{\top}$ refining each a (corresponding to the intersection of zero base sorts). Additionally, we assume that there is a judgment $r_1^a \overset{\text{def}^a}{\leq} r_2^a$ defined on pairs of base sorts that refine the same base type a . We omit the base type a from occurrences of $\overset{\text{def}^a}{\&}$, $\overset{\text{def}^a}{\top}$ and $\overset{\text{def}^a}{\leq}$ when the base type is clear from the context.

⁹If we had full β reduction here instead of β -value, we couldn't include variables as values, since then an expression might be substituted for a variable. This would violate subject reduction if intersection introduction was used for the variable.

¹⁰A comment on η -expansion might be appropriate here.

We further assume that $\overset{\text{def}^a}{\&}$, $\overset{\text{def}^a}{\top}$ and $\overset{\text{def}^a}{\leq}$ satisfy the following conditions (which are based on Freeman [Fre94]) :

Assumption 1 (base reflexivity)

For all r^a we have $r^a \overset{\text{def}}{\leq} r^a$.

Assumption 2 (base transitivity)

If $r_1^a \overset{\text{def}}{\leq} r_2^a$ and $r_2^a \overset{\text{def}}{\leq} r_3^a$ then $r_1^a \overset{\text{def}}{\leq} r_3^a$.

Assumption 3 ($\overset{\text{def}}{\&}$ is lower bound)

$r_1^a \overset{\text{def}}{\&} r_2^a \overset{\text{def}}{\leq} r_1^a$ and $r_1^a \overset{\text{def}}{\&} r_2^a \overset{\text{def}}{\leq} r_2^a$.

Assumption 4 ($\overset{\text{def}}{\&}$ is maximal)

If $s^a \overset{\text{def}}{\leq} r_1^a$ and $s^a \overset{\text{def}}{\leq} r_2^a$ then $s^a \overset{\text{def}}{\leq} r_1^a \overset{\text{def}}{\&} r_2^a$.

Assumption 5 ($\overset{\text{def}}{\top}$ is maximal)

$r^a \overset{\text{def}}{\leq} \overset{\text{def}^a}{\top}$.

Formally, these are assumptions regarding the existence of derivations of judgments of the form $r^a \overset{\text{def}}{\leq} s^a$. E.g. Assumption 2 can be written more formally as:

If there is a derivation of $r_1^a \overset{\text{def}}{\leq} r_2^a$ and also there is a derivation of $r_2^a \overset{\text{def}}{\leq} r_3^a$ then there is a derivation of $r_1^a \overset{\text{def}}{\leq} r_3^a$.

A concrete instance of base lattices is provided by the sort checker described in Chapter 7: it allows refinements of datatypes to be defined using an extension of regular tree grammars.

3.5 Refinement Restriction

The refinement judgment has the form:

$R \sqsubset A$ Sort R is a well-formed sort refining type A .

$$\frac{}{r^a \sqsubset a} \text{rf_base} \qquad \frac{R \sqsubset A \quad S \sqsubset B}{R \rightarrow S \sqsubset A \rightarrow B} \text{rf_arrow}$$

$$\frac{R \sqsubset A \quad S \sqsubset A}{R \& S \sqsubset A} \text{rf_conj} \qquad \frac{}{\top_A \sqsubset A} \text{rf_top}$$

A direct consequence of this definition is that every sort refines at most one type. We extend the refinement notation pointwise to $\Delta \sqsubset \Gamma$ for sort context Δ and type context Γ which bind the same variables. Note that the refinement relation is *not* a subtyping (or subsorting) relation and is therefore neither co- nor contra-variant in the usual sense. Subsoring is introduced in the next section.

We say that a sort R is *well-formed* if it refines some type A . In what follows, we will only be interested in well-formed sorts, and so when we use the term “sort” we implicitly mean “well-formed sort”. We say that two sorts are *compatible* if they refine the same type. We say that a sort S is a *conjunct* of sort R if $S = R$, or (inductively) if $R = R_1 \ \& \ R_2$ and S is a conjunct of either R_1 or R_2 .

3.6 Declarative Subsoring

The subsoring judgement has the form, where R and S must be compatible sorts:

$$R \leq S \quad \text{Sort } R \text{ is a subsort of } S.$$

It is defined by the following rules, which are standard for intersection types except for the omission of distributivity and the addition of subtyping for the base sorts.

$$\begin{array}{c}
\frac{r \stackrel{\text{def}}{\leq} s}{r \leq s} \text{sub_def} \quad \frac{}{r^a \& s^a \leq r^a \stackrel{\text{def}}{\&} s^a} \text{sub_inter_def} \\
\\
\frac{}{\top^a \leq \top} \text{sub_top_def} \\
\\
\frac{}{R \leq R} \text{sub_reflex} \quad \frac{R_1 \leq R_2 \quad R_2 \leq R_3}{R_1 \leq R_3} \text{sub_trans} \\
\\
\frac{}{R \& S \leq R} \text{sub_inter_left_1} \quad \frac{}{R \& S \leq S} \text{sub_inter_left_2} \\
\\
\frac{R \leq S_1 \quad R \leq S_2}{R \leq S_1 \& S_2} \text{sub_inter_right} \\
\\
\frac{}{R \leq \top^A} \text{sub_top} \\
\\
\frac{S_1 \leq R_1 \quad R_2 \leq S_2}{R_1 \rightarrow R_2 \leq S_1 \rightarrow S_2} \text{sub_arrow}
\end{array}$$

If $R \leq S$ and $S \leq R$ then we say R and S are *equivalent* sorts, and write $R \cong S$. \cong satisfies the usual properties for an equivalence: it is reflexive (from `sub_reflex`), transitive (from `sub_trans`) and symmetric (it has a symmetric definition).

Lemma 3.6.1 (\cong is a Congruence)

If $R_1 \cong S_1$ and $R_2 \cong S_2$ then $R_1 \rightarrow R_2 \cong S_1 \rightarrow S_2$ and $R_1 \& R_2 \cong S_1 \& S_2$.

Proof: Subsorting in each direction can be derived using `sub_arrow` for the first part, and the rules `sub_inter_left_1`, `sub_inter_left_2` and `sub_inter_right` for the second part. \square

Lemma 3.6.2 (Associativity and Commutivity of $\&$)

$R \& S \cong S \& R$ and $R_1 \& (R_2 \& R_3) \cong (R_1 \& R_2) \& R_3$.

Proof: Subtyping in each direction of each part can be derived using only the rules `sub_inter_left_1`, `sub_inter_left_2` and `sub_inter_right`. \square

Lemma 3.6.3 (Equality of defined and syntactic intersections)

1. $r^a \& s^a \stackrel{\text{def}}{\cong} r^a \& s^a$
2. $\top \stackrel{\text{def}^a}{\cong} \top^a$

Proof:

1. Rule `sub_inter_def` gives one direction, and the other direction is obtained by applying rule `sub_inter_right` to the two parts of Assumption 3.
2. Using `sub_top_def` and `sub_top`.

\square

It is our intention that whenever we have $R \cong S$ we can replace occurrences of R by S . Thus, we will often restrict attention to one representative of each equivalence class of sorts with respect to \cong .

3.7 Finiteness of refinements

We will now show that for each type there are only a finite number of equivalence classes of refinements. Our proof is quite different to that given by Freeman [FP91], which depends upon the existence of a suitable notion of “splitting” refinements, and is intended to yield a practical algorithm for enumerating the refinements of a type. Instead, we give a simple proof that requires minimal assumptions, and thus should easily extend to other specific instances of refinement types.

First we will need the following definition, which is unambiguous because of Lemma 3.6.2:

Definition 3.7.1

If $\{R_1, \dots, R_n\}$ is a finite set of sorts each refining type A then $\&\{R_1, \dots, R_n\}$ is defined to be the equivalence class containing $R_1 \& \dots \& R_n$, or \top^A if $n = 0$.

The following lemma captures our intention that the only refinements of base types are base sorts, up to equivalence:

Lemma 3.7.2 (Finiteness of Base Refinements)

For every sort R such that $R \sqsubset a$ for some base type a , there is a base sort s such that $R \cong s$.

Proof: By induction on the structure of R . We have two cases:

Case: $R = s$.

Rule `sub_reflex` gives $s \leq s$, thus $s \cong s$.

Case: $R = R_1 \& R_2$.

By the induction hypothesis, we have $R_1 \cong s_1^a$ and $R_2 \cong s_2^a$, i.e. there must be derivations:

$\mathcal{D}_1 :: R_1 \leq s_1^a$ and $\mathcal{D}'_1 :: s_1^a \leq R_1$

$\mathcal{D}_2 :: R_2 \leq s_2^a$ and $\mathcal{D}'_2 :: s_2^a \leq R_2$.

We then construct the derivations:

$$\frac{\frac{\frac{}{R_1 \& R_2 \leq R_1} \quad \mathcal{D}_1}{R_1 \leq s_1^a} \quad \frac{\frac{}{R_1 \& R_2 \leq R_2} \quad \mathcal{D}_2}{R_1 \leq s_2^a}}{R_1 \& R_2 \leq s_1^a \quad R_1 \& R_2 \leq s_2^a} \quad \frac{}{s_1^a \& s_2^a \leq s_1^a \text{ def } \& s_2^a}}{R_1 \& R_2 \leq s_1^a \& s_2^a} \quad \frac{}{s_1^a \& s_2^a \leq s_1^a \text{ def } \& s_2^a}}{R_1 \& R_2 \leq s_1^a \text{ def } \& s_2^a}$$

and

$$\frac{\frac{\frac{}{s_1^a \text{ def } \& s_2^a \text{ def } \leq s_1^a} \quad \mathcal{E}_1}{s_1^a \text{ def } \& s_2^a \leq s_1^a} \quad \mathcal{D}'_1}{s_1^a \text{ def } \& s_2^a \leq R_1} \quad \frac{\frac{\frac{}{s_1^a \text{ def } \& s_2^a \text{ def } \leq s_2^a} \quad \mathcal{E}_2}{s_1^a \text{ def } \& s_2^a \leq s_2^a} \quad \mathcal{D}'_2}{s_1^a \text{ def } \& s_2^a \leq R_2}}{s_1^a \text{ def } \& s_2^a \leq R_1 \quad s_1^a \text{ def } \& s_2^a \leq R_2} \quad \frac{}{s_1^a \text{ def } \& s_2^a \leq R_1 \& R_2}$$

Where \mathcal{E}_1 and \mathcal{E}_2 are the appropriate derivations given by Assumption 3.

□

A simple consequence of this lemma is that each base type has only a finite number of distinct refinements modulo sort equivalence. The following theorem extends this property to all types.

Theorem 3.7.3 (Finiteness of Refinements)

For each type A , there is a finite set Σ_A which contains a representative of each of the equivalence classes of refinements of A .

Proof: By induction on the structure of A .

Case: $A = a$.

Then each refinement of a is equivalent to one of the base sorts r^a (by Lemma 3.7.2), and there are only a finite number of these. Thus, we let Σ_a be the finite set containing all r^a .

Case: $A = A_1 \rightarrow A_2$.

By the induction hypothesis, we have finite sets $\Sigma_{A_1}, \Sigma_{A_2}$. We define:

$$\begin{aligned}\Psi &= \{R_1 \rightarrow R_2 \mid R_1 \in \Sigma_{A_1}, R_2 \in \Sigma_{A_2}\} \\ \Sigma'_{A_1 \rightarrow A_2} &= \{\& \Psi' \mid \Psi' \subset \Psi\}\end{aligned}$$

We then let $\Sigma_{A_1 \rightarrow A_2}$ contain one representative for each element of $\Sigma'_{A_1 \rightarrow A_2}$ (each of which is an equivalence class).

Ψ is finite, with size bounded by the product of the sizes of Σ_{A_1} and Σ_{A_2} . Thus, $\Sigma_{A_1 \rightarrow A_2}$ is finite with size bounded by $2^{(\text{size of } \Psi)}$.

It remains to show that every refinement R with $R \sqsubset A_1 \rightarrow A_2$ is in one of the equivalence classes in $\Sigma'_{A_1 \rightarrow A_2}$. We show this by a nested induction on R . We have three subcases:

Subcase: $R = R_1 \rightarrow R_2$.

By inversion $R_1 \sqsubset A_1$ and $R_2 \sqsubset A_2$.

Thus, R_1 and R_2 are equivalent to some $R'_1 \in \Sigma_{A_1}$ and $R'_2 \in \Sigma_{A_2}$ (using the first induction hypothesis).

Then $R_1 \rightarrow R_2 \cong R'_1 \rightarrow R'_2$ (by Lemma 3.6.1), and $R'_1 \rightarrow R'_2$ is in equivalence class $\&\{R'_1 \rightarrow R'_2\} \in \Sigma'_{A_1 \rightarrow A_2}$ (by the above definition of $\Sigma'_{A_1 \rightarrow A_2}$).

Subcase: $R = R_1 \& R_2$.

By inversion $R_1 \sqsubset A_1 \rightarrow A_2$ and $R_2 \sqsubset A_1 \rightarrow A_2$.

Thus by the second induction hypothesis, R_1 and R_2 are in equivalence classes $\&\Psi_1$ and $\&\Psi_2$ for some $\Psi_1 \subset \Psi$ and $\Psi_2 \subset \Psi$.

Finally, $R_1 \& R_2$ is in equivalence class $\&(\Psi_1 \cup \Psi_2)$ (using

Lemma 3.6.1 and Lemma 3.6.2), and $\&(\Psi_1 \cup \Psi_2) \in \Sigma'_{A_1 \rightarrow A_2}$ (by the definition of $\Sigma'_{A_1 \rightarrow A_2}$).

Subcase: $R = \top^{A_1 \rightarrow A_2}$.

Using the above definition of $\Sigma'_{A_1 \rightarrow A_2}$ and the definition of $\& \Psi'$ when Ψ' is the empty set.

□

3.8 Algorithmic Subsorting

The rules in Section 3.6 do not immediately yield an algorithm for deciding subsorting. We thus present the following algorithmic subsorting judgment, and show that it is equivalent to the declarative subsorting rules. Due to the absence of distributivity, our subsorting algorithm is quite different to previous algorithms proposed for intersection types, such as those by Reynolds [Rey88] and Freeman [Fre94]. For efficiency, our algorithm first simplifies the sorts that we wish to compare so that they do not contain any intersections of base sorts. A simplified sort must match the following grammar:

$$\begin{aligned} \text{Simplified Sorts } R^s & ::= r^a \mid R^f \\ \text{Simplified Function Sorts } R^f & ::= R_1^s \rightarrow R_2^s \mid R_1^f \& R_2^f \mid \top^{A \rightarrow B} \end{aligned}$$

The following function simplifies a well-formed sort R :

$$\begin{aligned} |r| & = r \\ |R \rightarrow S| & = |R| \rightarrow |S| \\ |R \& S| & = |R| \stackrel{\text{def}}{\&} |S| & \text{(if } R, S \sqsubset a) \\ |R \& S| & = |R| \& |S| & \text{(if } R, S \sqsubset A \rightarrow B) \\ |\top^a| & = \stackrel{\text{def}^a}{\top} \\ |\top^{A \rightarrow B}| & = \top^{A \rightarrow B} \end{aligned}$$

Lemma 3.8.1 (Correctness of simplification)

$|R|$ is a simplified sort satisfying $|R| \cong R$.

Proof: By induction on R .

Case: $R = r$.

Then $|r| = r$ which is a simplified sort satisfying $r \cong r$ (by reflexivity of \cong).

Case: $R = R_1 \rightarrow R_2$.

By the induction hypothesis on R_1 and R_2 . We use Lemma 3.6.1 for the first part. For the second part we use the definitions of R^f and R^s .

Case: $R = R_1 \& R_2$, with $R_1, R_2 \sqsubset a$.

By the induction hypothesis $|R_1| \cong R_1$ and $|R_2| \cong R_2$. For the first part we use Lemma 3.6.1, and then Lemma 3.6.3. For the second part, $|R_1|, |R_2|$ must have the forms r_1^a, r_2^a and then $r_1^a \& r_2^a$ is defined, and has the form s^a .

Case: $R = R_1 \& R_2$, with $R_1, R_2 \sqsubset A \rightarrow B$.

By the induction hypothesis on R_1 and R_2 . We use Lemma 3.6.1 for the first part. For the second part, the induction hypothesis implies that $|R_1|$ and $|R_2|$ are simplified sorts, and that they refine the same type $A \rightarrow B$ as R_1 and R_2 , so we can use inversion to determine that $|R_1|, |R_2|$ must be simplified function sorts.

Case: $R = \top^{A \rightarrow B}$. Similar to the case for $R = r$.

□

The core of our algorithm is given by the following judgment which relates two compatible simplified sorts:

$R^s \trianglelefteq S^s$ Simplified sort R^s is algorithmically a subsort of simplified sort S^s .

This judgment has the following rules. We often omit the superscripts s and f to avoid clutter when they can be easily reconstructed.

$$\begin{array}{c}
\frac{r \stackrel{\text{def}}{\trianglelefteq} s}{r \trianglelefteq s} \text{subalg_base} \quad \frac{S_1 \trianglelefteq R_1 \quad R_2 \trianglelefteq S_2}{R_1 \rightarrow R_2 \trianglelefteq S_1 \rightarrow S_2} \text{subalg_arrow} \\
\\
\frac{R_1 \trianglelefteq S_1 \rightarrow S_2}{R_1 \& R_2 \trianglelefteq S_1 \rightarrow S_2} \text{subalg_interL1} \quad \frac{R_2 \trianglelefteq S_1 \rightarrow S_2}{R_1 \& R_2 \trianglelefteq S_1 \rightarrow S_2} \text{subalg_interL2} \\
\\
\frac{R^f \trianglelefteq S_1^f \quad R^f \trianglelefteq S_2^f}{R^f \trianglelefteq S_1^f \& S_2^f} \text{subalg_interR} \quad \frac{}{R \trianglelefteq \top^{A \rightarrow B}} \text{subalg_topR}
\end{array}$$

We now prove some simple lemmas needed to show that algorithmic and declarative subtyping coincide.

Lemma 3.8.2 (Monotonicity of Algorithmic Subtyping)

If $R_1^f \trianglelefteq S^f$ then $R_1^f \& R_2^f \trianglelefteq S^f$ and $R_1^f \& R_2^f \trianglelefteq S^f$.

Proof: By induction on S^f .

Case: $S^f = S_1^s \rightarrow S_2^s$.

By `subalg_interL1` (first part) and `subalg_interL2` (second part).

Case: $S^f = S_1^f \& S_2^f$.

Let $\mathcal{D} :: R_1^f \trianglelefteq S_1^f \& S_2^f$ be the given derivation.

$$\text{By inversion, } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{R^f \trianglelefteq S_1^f} \quad \frac{\mathcal{D}_2}{R \trianglelefteq S_2^f}}{R^f \trianglelefteq S_1^f \& S_2^f} \text{subalg_interR}$$

Applying the induction hypothesis to \mathcal{D}_1 and \mathcal{D}_2 and then using rule `subalg_interR` yields the required result.

Case: $S^f = \top_{A \rightarrow B}$.

Both parts are immediate, using rule `subalg_topR`.

□

Lemma 3.8.3 (Reflexivity of Algorithmic Subtyping)

$R^s \trianglelefteq R^s$.

Proof: By induction on R^s .

Case: $R^s = r^a$.

By Assumption 1 and rule `sub_def`.

Case: $R^s = R_1^s \rightarrow R_2^s$.

The induction hypothesis yields $R_1^s \trianglelefteq R_1^s$ and $R_2^s \trianglelefteq R_2^s$. Applying rule `subalg_arrow` yields the required result.

¹¹There's a choice here between working with all sorts, or requiring that intersections of base sorts be simplified via $\stackrel{\text{def}}{\&}$. I've chosen the later, since this is what the implementation does.

Case: $R^s = R_1^f \& R_2^f$.

Using the induction hypothesis on R_1^f and R_2^f , then Lemma 3.8.2 on each part, then rule `subalg_interR`.

Case: $R^s = \top^{A \rightarrow B}$.

By rule `subalg_topR`.

□

Lemma 3.8.4 (Transitivity of Algorithmic Subtyping)

If $R_1^s \sqsubseteq R_2^s$ and $R_2^s \sqsubseteq R_3^s$ then $R_1^s \sqsubseteq R_3^s$.

Proof: By induction on R_2^s and the derivations $\mathcal{D}_2 :: R_2^s \sqsubseteq R_3^s$ and $\mathcal{D}_1 :: R_1^s \sqsubseteq R_2^s$, ordered lexicographically. We have the following cases for \mathcal{D}_2 :

Case: $\mathcal{D}_2 = \frac{\overset{\text{def}}{r_2 \leq r_3}}{r_2 \sqsubseteq r_3} \text{subalg_base}$.

\mathcal{D}_1 must have the form $\frac{\overset{\text{def}}{r_1 \leq r_2}}{r_1 \sqsubseteq r_2} \text{subalg_base}$

and then we use transitivity of $\overset{\text{def}}{\leq}$ (Assumption 2).

Case: $\mathcal{D}_2 = \frac{\mathcal{D}_{21} \quad \mathcal{D}_{22}}{R_2 \sqsubseteq S_1 \& S_2} \text{subalg_interR}$.

Applying the induction hypothesis to the pairs $\mathcal{D}_1, \mathcal{D}_{21}$ and $\mathcal{D}_1, \mathcal{D}_{22}$ yields the derivations $\mathcal{D}_{31} :: R_1 \sqsubseteq S_1$ and $\mathcal{D}_{32} :: R_1 \sqsubseteq S_2$, to which we apply the rule `subalg_interR`.

Case: $\mathcal{D}_2 = \frac{\mathcal{D}'_2}{R_{21} \& R_{22} \sqsubseteq S_1 \rightarrow S_2} \text{subalg_interL1}$

Then \mathcal{D}_1 must have the form $\frac{\mathcal{D}_{11} \quad \mathcal{D}_{12}}{R_1 \sqsubseteq R_{21} \& R_{22}} \text{subalg_interR}$.

Applying the induction hypothesis to \mathcal{D}_{11} and \mathcal{D}'_2 yields the required result.

$$\text{Case: } \mathcal{D}_2 = \frac{\mathcal{D}'_2}{R_{21} \& R_{22} \trianglelefteq S_1 \rightarrow S_2} \text{subalg_interL2.}$$

Symmetric to the previous case.

$$\text{Case: } \mathcal{D}_2 = \frac{}{R_2 \trianglelefteq \top^{A \rightarrow B}} \text{subalg_topR.}$$

Immediate, using rule `subalg_topR`.

$$\text{Case: } \mathcal{D}_2 = \frac{\begin{array}{c} \mathcal{D}_{21} \\ R_{31} \trianglelefteq R_{21} \end{array} \quad \begin{array}{c} \mathcal{D}_{22} \\ R_{22} \trianglelefteq R_{32} \end{array}}{R_{21} \rightarrow R_{22} \trianglelefteq R_{31} \rightarrow R_{32}} \text{subalg_arrow.}$$

We have the following subcases for \mathcal{D}_1 :

$$\text{Subcase: } \mathcal{D}_1 = \frac{\mathcal{D}'_1}{R_{11} \trianglelefteq R_{21} \rightarrow R_{22}} \text{subalg_interL1.}$$

Applying the induction hypothesis to \mathcal{D}'_1 and \mathcal{D}_2 yields a derivation of $R_{11} \trianglelefteq R_{31} \rightarrow R_{32}$, to which we apply rule `subalg_interL1`.

$$\text{Subcase: } \mathcal{D}_1 = \frac{\mathcal{D}'_1}{R_{11} \& R_{12} \trianglelefteq R_{21} \rightarrow R_{22}} \text{subalg_interL2.}$$

Symmetric to the previous subcase.

$$\text{Subcase: } \mathcal{D}_1 = \frac{\begin{array}{c} \mathcal{D}_{11} \\ R_{21} \trianglelefteq R_{11} \end{array} \quad \begin{array}{c} \mathcal{D}_{12} \\ R_{12} \trianglelefteq R_{22} \end{array}}{R_{11} \rightarrow R_{12} \trianglelefteq R_{21} \rightarrow R_{22}} \text{subalg_arrow.}$$

By the induction hypothesis on $R_{21}, \mathcal{D}_{21}, \mathcal{D}_{11}$ and $R_{22}, \mathcal{D}_{12}, \mathcal{D}_{22}$ we have $R_{31} \trianglelefteq R_{11}$ and $R_{12} \trianglelefteq R_{32}$. Applying rule `subalg_arrow` yields the required result.

□

The following theorem demonstrates that the core of our subsorting algorithm is correct, i.e that it coincides with the declarative formulation of subsorting on simplified sorts.

Theorem 3.8.5 (Correctness of \trianglelefteq) $R^s \trianglelefteq S^s$ if and only if $R^s \leq S^s$.

Proof: We first show the “only if” part by showing that if there is a derivation $\mathcal{D} :: R^s \trianglelefteq S^s$ then there is a derivation $\mathcal{E} :: R^s \leq S^s$, by induction on \mathcal{D} . We have the following cases for \mathcal{D} :

$$\text{Case: } \mathcal{D} = \frac{r \stackrel{\text{def}}{\leq} s}{r \trianglelefteq s} \text{ subalg_base.}$$

$$\text{Construct } \mathcal{E} = \frac{r \stackrel{\text{def}}{\leq} s}{r \leq s} \text{ sub_def.}$$

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ S_1 \trianglelefteq R_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ R_2 \trianglelefteq S_2 \end{array}}{R_1 \rightarrow R_2 \trianglelefteq S_1 \rightarrow S_2} \text{ subalg_arrow.}$$

$$\text{Construct } \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ S_1 \leq R_1 \end{array} \quad \begin{array}{c} \mathcal{E}_2 \\ R_2 \leq S_2 \end{array}}{R_1 \rightarrow R_2 \leq S_1 \rightarrow S_2} \text{ sub_arrow} \quad ^{12}$$

where \mathcal{E}_1 and \mathcal{E}_2 are obtained by applying the induction hypothesis to \mathcal{D}_1 and \mathcal{D}_2 .

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ R_1 \trianglelefteq S_1 \rightarrow S_2 \end{array}}{R_1 \ \& \ R_2 \trianglelefteq S_1 \rightarrow S_2} \text{ subalg_interL1.}$$

We apply the induction hypothesis to \mathcal{D}_1 to yield \mathcal{E}_1 and then construct the derivation:

$$\frac{\frac{}{R_1 \ \& \ R_2 \trianglelefteq R_1} \text{ sub_inter_left_1} \quad \begin{array}{c} \mathcal{E}_1 \\ R_1 \leq S_1 \rightarrow S_2 \end{array}}{R_1 \ \& \ R_2 \trianglelefteq S_1 \rightarrow S_2} \text{ sub_trans.}$$

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ R_2 \trianglelefteq S_1 \rightarrow S_2 \end{array}}{R_1 \ \& \ R_2 \trianglelefteq S_1 \rightarrow S_2} \text{ subalg_interL2.}$$

Symmetric to the previous case.

¹² Too much detail here?

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{R \trianglelefteq S_1} \quad \frac{\mathcal{D}_2}{R \trianglelefteq S_2}}{R \trianglelefteq S_1 \ \& \ S_2} \text{subalg_interR.}$$

We apply the induction hypothesis to \mathcal{D}_1 and \mathcal{D}_2 and then use rule `sub_inter_right`.

$$\text{Case: } \mathcal{D} = \frac{}{R \trianglelefteq \top^{A \rightarrow B}} \text{subalg_topR.}$$

By rule `sub_top`.

We now show the “if” part of the theorem by showing that if there is a derivation $\mathcal{E} :: R^s \leq S^s$ then there is a derivation $\mathcal{D} :: R^s \trianglelefteq S^s$, by induction on \mathcal{E} . We have the following cases for \mathcal{E} :

$$\text{Case: } \mathcal{E} = \frac{r \stackrel{\text{def}}{\leq} s}{r \leq s} \text{sub_def.}$$

$$\text{Construct } \mathcal{D} = \frac{r \stackrel{\text{def}}{\leq} s}{r \trianglelefteq s} \text{subalg_base.}$$

$$\text{Case: } \mathcal{E} = \frac{}{r^a \ \& \ s^a \leq r^a \stackrel{\text{def}}{\&} \ s^a} \text{sub_inter_def.}$$

Outside the scope of the theorem, since $r^a \ \& \ s^a$ is not a simplified sort.

$$\text{Case: } \mathcal{E} = \frac{}{R^s \leq R^s} \text{sub_reflex.}$$

Using Lemma 3.8.3.

$$\text{Case: } \mathcal{E} = \frac{\frac{\mathcal{E}_1}{R_1 \leq R_2} \quad \frac{\mathcal{E}_2}{R_2 \leq R_3}}{R_1 \leq R_3} \text{sub_trans.}$$

By applying the induction hypothesis to \mathcal{E}_1 and \mathcal{E}_2 and then using Lemma 3.8.4.

$$\text{Case: } \mathcal{E} = \frac{}{R_1 \ \& \ R_2 \leq R_1} \text{sub_inter_left_1.}$$

Since R_1 & R_2 must be a simplified sort, R_1 and R_2 must be simplified function sorts. We then use Lemma 3.8.3 to obtain a derivation of $R_1 \sqsubseteq R_1$ and then apply Lemma 3.8.2.

Case: $\mathcal{E} = \frac{}{R_1 \ \& \ R_2 \leq R_2}$ `sub_inter_left_2`.

Symmetric to the previous case.

Case: $\mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ R \leq S_1 \end{array} \quad \begin{array}{c} \mathcal{E}_2 \\ R \leq S_2 \end{array}}{R \leq S_1 \ \& \ S_2}$ `sub_inter_right`.

Since S_1 & S_2 must be a simplified sort, S_1 and S_2 must be simplified function sorts. We apply the induction hypothesis to \mathcal{E}_1 and \mathcal{E}_2 and then apply rule `subalg_interR`.

Case: $\mathcal{E} = \frac{}{R \leq \top^A}$ `sub_top`.

Since \top^A must be a simplified sort, we have $A = A_1 \rightarrow A_2$. We then use rule `subalg_topR`.

Case: $\mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ S_1 \leq R_1 \end{array} \quad \begin{array}{c} \mathcal{E}_2 \\ R_2 \leq S_2 \end{array}}{R_1 \rightarrow R_2 \leq S_1 \rightarrow S_2}$ `sub_arrow`.

Construct $\mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ S_1 \sqsubseteq R_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ R_2 \sqsubseteq S_2 \end{array}}{R_1 \rightarrow R_2 \sqsubseteq S_1 \rightarrow S_2}$ `subalg_arrow`

where \mathcal{D}_1 and \mathcal{D}_2 are obtained by applying the induction hypothesis to \mathcal{E}_1 and \mathcal{E}_2 .¹³

□

The following theorem demonstrates that we can construct a correct algorithm for determining subsorting by simplifying sorts and then comparing them using \sqsubseteq .

¹³*Too much detail here?*

Theorem 3.8.6 (Correctness of Algorithmic Subtyping)

$R \leq S$ if and only if $|R| \trianglelefteq |S|$.

Proof: $|R| \cong R$ and $|S| \cong S$ (first part of Lemma 3.8.1), so $R \leq S$ if and only if $|R| \leq |S|$ (using `sub.trans` and the definition of \cong). Further, $|R|$ and $|S|$ are simplified sorts (second part of Lemma 3.8.1). We then use the above theorem. \square

As well as demonstrating the correctness of an algorithm for determining subtyping, we will also make use of this theorem in later proofs to convert between the declarative and algorithmic forms of subtyping. In particular, we will often convert to the algorithmic form to reduce the number of cases that we need to consider.

3.9 Declarative Sorting

The sorting judgment for terms relates a term with type A in a type context Γ to some of the refinements of A under suitable sort contexts refining Γ .

$\Delta \vdash_{\Sigma} M \in R$ Term M has sort R in sort context Δ under signature Σ .

The sorting rules are standard for systems with intersection types, with the exception that the introduction rule for intersections is restricted to values.

$$\begin{array}{c}
\frac{x \in R \text{ in } \Delta}{\Delta \vdash_{\Sigma} x \in R} \text{sort_var} \qquad \frac{c \in R \text{ in } \Sigma}{\Delta \vdash_{\Sigma} c \in R} \text{sort_const} \\
\\
\frac{\Delta, x \in R \vdash_{\Sigma} M \in S}{\Delta \vdash_{\Sigma} \lambda x:A. M \in R \rightarrow S} \text{sort_lam} \\
\\
\frac{\Delta \vdash_{\Sigma} M \in R \rightarrow S \quad \Delta \vdash_{\Sigma} N \in R}{\Delta \vdash_{\Sigma} MN \in S} \text{sort_app} \\
\\
\frac{\Delta \vdash_{\Sigma} V \in R \quad \Delta \vdash_{\Sigma} V \in S}{\Delta \vdash_{\Sigma} V \in R \ \& \ S} \text{sort_inter} \qquad \frac{}{\Delta \vdash_{\Sigma} V \in T^A} \text{sort_top} \\
\\
\frac{\Delta \vdash_{\Sigma} M \in R \quad R \leq S}{\Delta \vdash_{\Sigma} M \in S} \text{sort_subs}
\end{array}$$

In order to demonstrate that these rules are sensible we demonstrate that they satisfy some standard properties. The most important is Subject Reduction, namely that β -reduction preserves sorts. But first we need to show some structural properties: weakening exchange, contraction and substitution. These are standard properties for hypothetical judgments (see e.g. [Pfe01]): in this case our hypotheses are the assumed sorts for variables in the sort context.

Lemma 3.9.1 (Weakening, Exchange, Contraction)

1. If $\Delta \vdash_{\Sigma} M \in R$ then $\Delta, x \in S \vdash_{\Sigma} M \in R$.
2. If $\Delta, x \in S_1, y \in S_2, \Delta' \vdash_{\Sigma} M \in R$ then $\Delta, y \in S_2, x \in S_1, \Delta' \vdash_{\Sigma} M \in R$.
3. If $\Delta, x \in S, y \in S, \Delta' \vdash_{\Sigma} M \in R$ then $\Delta, w \in S, \Delta' \vdash_{\Sigma} \{w/x\}\{w/y\}M \in R$.

Proof: In each case by induction over the structure of the given sorting derivation. The cases for the rule `sort_var` are straightforward; the cases for other rules simply follow the structure of the given derivation. \square

Lemma 3.9.2 (Value Preservation)

1. $[V'/x]V$ is a value.

Proof: By straightforward inductions on V . □

Lemma 3.9.3 (Substitution Lemma)

If $\Delta \vdash_{\Sigma} V \in R$ and $\Delta, x \in R \vdash N \in S$ then $\Delta \vdash_{\Sigma} \{V/x\}N \in S$.

Proof:

Let $\mathcal{D}_1 :: \Delta \vdash_{\Sigma} V \in R$ and $\mathcal{D}_2 :: \Delta, x \in R \vdash N \in S$ be the given derivations. The proof is by a simple induction on \mathcal{D}_2 , constructing the derivation $\mathcal{D}_3 :: \Delta \vdash_{\Sigma} \{V/x\}N \in S$. We show three interesting cases:

$$\text{Case: } \mathcal{D}_2 = \frac{x \in R \text{ in } \Delta, x \in R}{\Delta, x \in R \vdash_{\Sigma} x \in R} \text{sort_var.}$$

Then $R = S$ and $N = x$ thus $\{V/x\}N = V$ and so we simply use $\mathcal{D}_3 = \mathcal{D}_1 :: \Delta \vdash_{\Sigma} V \in R$.

$$\text{Case: } \mathcal{D}_2 = \frac{y \in S \text{ in } \Delta, x \in R}{\Delta, x \in R \vdash_{\Sigma} y \in S} \text{sort_var} \text{ with } y \neq x.$$

Then $\{V/x\}y = y$ and $y \in S$ is in Δ so we can simply use the variable rule:

$$\mathcal{D}_3 = \frac{y \in S \text{ in } \Delta}{\Delta \vdash_{\Sigma} y \in S} \text{sort_var.}$$

$$\text{Case: } \mathcal{D}_2 = \frac{\frac{\mathcal{D}_{21}}{\Delta, x \in R \vdash_{\Sigma} V' \in S_1} \quad \frac{\mathcal{D}_{22}}{\Delta, x \in R \vdash_{\Sigma} V' \in S_2}}{\Delta, x \in R \vdash_{\Sigma} V' \in R \ \& \ S} \text{sort_inter.}$$

Applying the induction hypothesis to \mathcal{D}_{21} and \mathcal{D}_{22} yields the derivations $\mathcal{D}_{31} :: \Delta \vdash_{\Sigma} \{V/x\}V' \in S_1$ and $\mathcal{D}_{32} :: \Delta \vdash_{\Sigma} \{V/x\}V' \in S_2$.

Since $\{V/x\}V'$ is a value (by the value preservation lemma above) we can apply rule `sort_inter` to these derivations to obtain $\mathcal{D}_3 :: \Delta \vdash_{\Sigma} \{V/x\}V' \in S_1 \ \& \ S_2$, as required.

The remaining cases simply reconstruct the derivation \mathcal{D}_3 following the structure of \mathcal{D}_2 , similar to the case for `sort_inter` above. □

This proof is relatively independent of the definition of values: it only requires that the substitution $\{V/x\}V'$ always yields a value. Thus, a similar result holds for alternative definitions of values that satisfy this criteria. E.g. we could define values to include all terms, so that the “value” restriction in rule `sort_inter` becomes vacuous. Then the above lemma would be the substitution lemma for a system without distributivity but with no value restriction. Such alternative definitions are not the main focus of this dissertation, but we will occasionally consider them where they are particularly interesting or where we feel that they demonstrate the robustness of our results.

A simple corollary of the Substitution Lemma is obtained by considering the case where V is a variable:

Corollary 3.9.4 (Variable Subsumption)

If $\Delta, x \in R \vdash_{\Sigma} M \in S$ and $R' \leq R$ then $\Delta, x \in R' \vdash_{\Sigma} M \in S$.

Proof:

$\Delta, x \in R \vdash_{\Sigma} M \in S$	By assumption
$\Delta, y \in R', x \in R \vdash_{\Sigma} M \in S$	By weakening (Lemma 3.9.1 part 1)
(where y is not in $\Delta, x \in R$)	
$\Delta, y \in R' \vdash_{\Sigma} y \in R'$	By rule <code>sort_var</code>
$R' \leq R$	By assumption
$\Delta, y \in R' \vdash_{\Sigma} y \in R$	By rule <code>sort_subs</code>
$\Delta, y \in R' \vdash_{\Sigma} \{y/x\}M \in S$	By the Substitution Lemma (3.9.3)
$\Delta, x \in R' \vdash_{\Sigma} M \in S$	Replacing y by x
	(x is not in $\Delta, y \in R'$ nor in $\{y/x\}M$) □

[This result also holds with an alternative definition of values that excludes variables, although then it must be proved separately.]

The following property is critical to our proof of subject reduction. It generalizes similar properties in languages without subtyping or intersections.

Lemma 3.9.5 (Inversion for \rightarrow)

If $\Delta \vdash_{\Sigma} \lambda x:A_1. M \in R$ and $R \leq S_1 \rightarrow S_2$ then $\Delta, x \in S_1 \vdash_{\Sigma} M \in S_2$.

Proof: By induction on the derivation of the first assumption, $\mathcal{D} :: \Delta \vdash_{\Sigma} \lambda x:A_1. M \in R$. There are three possible cases for \mathcal{D} .

$$\text{Case: } \mathcal{D} = \frac{\mathcal{D}_1 \quad \Delta, x \in R_1 \vdash_{\Sigma} M \in R_2}{\Delta \vdash_{\Sigma} \lambda x:A. M \in R_1 \rightarrow R_2} \text{sort_lam.}$$

Then $R = R_1 \rightarrow R_2$ and so $R_1 \rightarrow R_2 \leq S_1 \rightarrow S_2$ (by assumption).
But then there must be a derivation $\mathcal{E} :: |R_1| \rightarrow |R_2| \trianglelefteq |S_1| \rightarrow |S_2|$
(by Correctness of Algorithmic Subsorting, Theorem 3.8.6).

$$\text{By inversion } \mathcal{E} = \frac{\mathcal{E}_1 \quad \mathcal{E}_2 \quad |S_1| \trianglelefteq |R_1| \quad |R_2| \trianglelefteq |S_2|}{|R_1| \rightarrow |R_2| \trianglelefteq |S_1| \rightarrow |S_2|} \text{subalg_arrow.}$$

Thus $S_1 \leq R_1$ and $R_2 \leq S_2$ (also by Correctness of Algorithmic Subsorting).

So $\Delta, x \in R_1 \vdash_{\Sigma} M \in S_2$ (by rule `sort_subs`)
and finally $\Delta, x \in S_1 \vdash_{\Sigma} M \in S_2$ (by Variable Subsumption, Corollary 3.9.4).

$$\text{Case: } \mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{E} \quad \Delta \vdash_{\Sigma} M \in R' \quad R' \leq R}{\Delta \vdash_{\Sigma} M \in R} \text{sort_subs.}$$

Then $R' \leq S_1 \rightarrow S_2$ (By rule `sub_trans`), so we can apply the induction hypothesis to \mathcal{D}_1 to obtain $\Delta, x \in S_1 \vdash_{\Sigma} M \in S_2$, as required.

$$\text{Case: } \mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \Delta \vdash_{\Sigma} V \in R_1 \quad \Delta \vdash_{\Sigma} V \in R_2}{\Delta \vdash_{\Sigma} V \in R_1 \ \& \ R_2} \text{sort_inter.}$$

Then $R = R_1 \ \& \ R_2$ and so $R_1 \ \& \ R_2 \leq S_1 \rightarrow S_2$ (by assumption).
Applying inversion to the corresponding algorithmic subsorting derivation (via the Correctness of Algorithmic Subsorting in the same way as the case for rule `sort_lam`) we find that we must have one of the following subcases:

Subcase: $R_1 \leq S_1 \rightarrow S_2$ (corresponding to rule `subalg_interL1`).

Then we apply the induction hypothesis to \mathcal{D}_1 , yielding $\Delta, x \in S_1 \vdash_{\Sigma} M \in S_2$, as required.

Subcase: $R_2 \leq S_1 \rightarrow S_2$ (corresponding to rule `subalg_interL2`).

Symmetric to the previous subcase.

□

The last case of this proof depends upon the strong inversion properties obtained from the algorithmic form of subsorting, in particular that $R_1 \& R_2 \leq S_1 \rightarrow S_2$ only if $R_1 \leq S_1 \rightarrow S_2$ or $R_2 \leq S_1 \rightarrow S_2$. This property fails if we allow the distributivity rule.¹⁴

We are now in a position to prove the main theorem of this subsection.

Theorem 3.9.6 (Sort Subject Reduction)

If $\Delta \vdash_{\Sigma} M \in R$ and $M \mapsto N$ then $\Delta \vdash_{\Sigma} N \in R$.

Proof: By induction on the structure of the derivation $\mathcal{D} :: \Delta \vdash_{\Sigma} M \in R$. We have the following cases for \mathcal{D} and the derivation $\mathcal{E} :: M \mapsto N$. We first treat the three sorting rules which do not correspond to term constructs, and then consider the remaining cases by inversion on \mathcal{E} .

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{\Delta \vdash_{\Sigma} V \in R_1} \quad \frac{\mathcal{D}_2}{\Delta \vdash_{\Sigma} V \in R_2}}{\Delta \vdash_{\Sigma} V \in R_1 \& R_2} \text{sort_inter.}$$

Then $M = V$ and by inversion the derivation of $M \mapsto N$ must be by rule `reduce_lam`, so $M = \lambda x:A. M'$ and $N = \lambda x:A. N'$.

By the induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2 we have $\Delta \vdash_{\Sigma} \lambda x:A. N' \in R_1$ and $\Delta \vdash_{\Sigma} \lambda x:A. N' \in R_2$. We then apply rule `sort_inter`.

$$\text{Case: } \mathcal{D} = \frac{}{\Delta \vdash_{\Sigma} V \in \top^A} \text{sort_top.}$$

Then $M = V$ and so $N = \lambda x:A. N'$ (as in the previous case) which is a value. Thus $\Delta \vdash_{\Sigma} N \in \top^A$ (by rule `sort_top`).

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{\Delta \vdash_{\Sigma} M \in R_1} \quad R_1 \leq R}{\Delta \vdash_{\Sigma} M \in R} \text{sort_subs.}$$

Then $\Delta \vdash_{\Sigma} N \in R_1$ (by the induction hypothesis on \mathcal{D}_1), thus $\Delta \vdash_{\Sigma} N \in R$ (by rule `sort_subs`).

¹⁴But there is a similar, but more complicated way of proving this inversion lemma with the subsorting algorithm in my thesis proposal, via the Lemma: If $R \cdot S_1 \gg R'_2$ and $\Delta \vdash_{\Sigma} M \in R$ then $\Delta \vdash_{\Sigma} M \in S_1 \rightarrow R'_2$.

$$\text{Case: } \mathcal{E} = \frac{\mathcal{E}_2 \quad M \mapsto N}{\lambda x:A. M \mapsto \lambda x:A. N} \text{ reduce_lam}$$

and

$$\mathcal{D} = \frac{\mathcal{D}_2 \quad \Delta, x \in R_1 \vdash_{\Sigma} M \in R_2}{\Delta \vdash_{\Sigma} \lambda x:A. M \in R_1 \rightarrow R_2} \text{ sort_lam.}$$

By the induction hypothesis on $\mathcal{D}_2, \mathcal{E}_2$ and then using rule `sort_lam`.

Case: \mathcal{E} is an instance of `reduce_app1` or `reduce_app2`. Similar to the previous case.

$$\text{Case: } \mathcal{E} = \frac{}{(\lambda x:A. M_1) V \mapsto [V/x]M_1} \beta_v$$

and

$$\mathcal{D} = \frac{\Delta \vdash_{\Sigma} \lambda x:A. M_1 \in R_2 \rightarrow R \quad \Delta \vdash_{\Sigma} V \in R_2}{\Delta \vdash_{\Sigma} (\lambda x:A. M_1) V \in R} \text{ sort_app.}$$

Then we apply the Inversion Lemma for \rightarrow (Lemma 3.9.5) to \mathcal{D}_1 to obtain

$$\Delta, x \in R_2 \vdash_{\Sigma} M_1 \in R$$

and then we apply the Substitution Lemma (Lemma 3.9.3), using \mathcal{D}_2 , to show

$$\Delta \vdash_{\Sigma} [V/x]M_1 \in R$$

as required. □

This theorem demonstrates that our calculus and sorting rules are at least basically sensible and internally consistent. In Chapter 4 we use similar proof techniques to show a corresponding sort preservation result for a functional language with reference cells. ¹⁵

¹⁵This proof is somewhat robust to changes in the definition of values, although it does require that if $V \mapsto N$ then N is a value.

3.10 Principal Sorts and Decidability of Inference

Two important properties of Standard ML are the existence of principal type schemes and a practical algorithm for finding them. We now consider the corresponding properties for the sorts of $\lambda^{v \rightarrow \&}$. We find that principal sorts exist, and that there is an algorithm for finding them, but both results seem to be mostly of theoretical interest. This is because both results require enumeration of all refinements of a type, which does not appear to be practical. The work by Freeman [Fre94] and experience with the implementation described later in this dissertation suggests that the exponential bound on the number of refinements in the proof of Theorem 3.7.3 (Finiteness of Refinements) is an accurate reflection the huge number of unique refinements at higher-order types. Regardless, these theorems and proofs are still interesting, if only to demonstrate where the enumeration is required.

The principal sorts theorem applies only to values: other terms do not necessarily have principal sorts. This is consistent with Standard ML, which has principal type schemes for values, but not for all expressions. E.g. the Standard ML expression `ref nil` can be assigned the type `(int list) ref` and also the type `(bool list) ref` but it can not be assigned the generalized type scheme `('a list) ref` (see [MTHM97]).

Theorem 3.10.1 (Principal Sorts)

If $\Gamma \vdash_{\Sigma} V : A$ and $\Delta \sqsubset \Gamma$ then there is some R_1 such that $R_1 \sqsubset A$ and $\Delta \vdash_{\Sigma} V \in R_1$ and for all R_2 such that $R_2 \sqsubset A$ and $\Delta \vdash_{\Sigma} V \in R_2$ we have $R_1 \leq R_2$.

Proof: We make use of the finite set Σ_A from Theorem 3.7.3 which contains representatives of all equivalence classes of refinements of A .

We construct the finite set

$$\Sigma = \{S_1 \in \Sigma_A \mid \Delta \vdash_{\Sigma} V \in S_1\}$$

and then we choose

$$R_1 = \& \Sigma.$$

This satisfies the first requirement of the theorem: $\Delta \vdash_{\Sigma} V \in R_1$ (by rule `sort_top` and repeated use of rule `sort_inter` following the structure of R_1).

It also satisfies the second requirement of the theorem: if $\Delta \vdash_{\Sigma} V \in R_2$ then R_2 is equivalent to some $S_2 \in \Sigma$ and so $R_1 \leq R_2$ (by transitivity and repeated use of rules `sub_inter_left1` and `sub_inter_left2`). \square

This proof is not constructive: it does not directly yield an algorithm for inferring principal sorts, because it does not specify a method for checking $\Delta \vdash_{\Sigma} V \in S_1$ for each S_1 . We now show that this problem is decidable. Our proof makes use of the finite set of representative refinements Σ_A constructed for each type A in Theorem 3.7.3 (Finiteness of Refinements). These sets can be impractically large, and so the algorithm in our proof is far from practical. Further, the previous proof also uses Σ_A which suggests that the principal sorts themselves might be too large to be practical.¹⁶

We will need the following inversion lemmas.

Lemma 3.10.2 (Inversion for Applications)

$\Delta \vdash_{\Sigma} M_1 M_2 \in R$ if and only if there is a sort S such that $\Delta \vdash_{\Sigma} M_1 \in S \rightarrow R$ and $\Delta \vdash_{\Sigma} M_2 \in S$.

Proof:

“If” part: By rule `sort_app`.

“Only if” part: By induction on the structure of the assumed derivation $\mathcal{D} :: \Delta \vdash_{\Sigma} M_1 M_2 \in R$. We have the following cases.

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} M_1 \in R_2 \rightarrow R \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} M_2 \in R_2 \end{array}}{\Delta \vdash_{\Sigma} M_1 M_2 \in R} \text{sort_app.}$$

Then we let $S = R_2$, and use \mathcal{D}_1 and \mathcal{D}_2 .

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}' \\ \Delta \vdash_{\Sigma} M_1 M_2 \in R' \end{array} \quad R' \leq R}{\Delta \vdash_{\Sigma} M_1 M_2 \in R} \text{sort_subs.}$$

Then there exists $S' \leq R'$ such that $\Delta \vdash_{\Sigma} M_1 \in S' \rightarrow R$ and $\Delta \vdash_{\Sigma} M_2 \in S'$ (by the induction hypothesis on \mathcal{D}'). But $S' \leq R$ (by rule `sub_trans`) and so we have the required result by choosing $S = S'$.

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} M_1 M_2 \in R_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} M_1 M_2 \in R_2 \end{array}}{\Delta \vdash_{\Sigma} M_1 M_2 \in R_1 \& R_2} \text{sort_inter.}$$

Cannot occur due to the value restriction on rule `sort_inter`, since

¹⁶Move the principal sorts theorem after the decidability result?

$M_1 M_2$ is not a value.

[With an alternative definition of values that includes applications, this case can be shown by applying the induction hypothesis to both \mathcal{D}_1 and \mathcal{D}_2 .]

Case: $\mathcal{D} = \frac{}{\Delta \vdash_{\Sigma} M_1 M_2 \in \top^A} \text{sort_top.}$

Cannot occur, as for the previous case.

□

Lemma 3.10.3 (Inversion for Variables)

$\Delta \vdash_{\Sigma} x \in R$ if and only if there is some S such that $x \in S$ is in Δ and $S \leq R$.

Proof:

“If” part: We simply construct the derivation

$$\frac{\frac{x \in S \text{ in } \Delta}{\Delta \vdash_{\Sigma} x \in S} \text{sort_var} \quad S \leq R}{\Delta \vdash_{\Sigma} x \in R} \text{sort_subs.}$$

“Only if” part: By induction on the structure of the assumed derivation $\mathcal{D} :: \Delta \vdash_{\Sigma} x \in R$. We have the following cases.

Case: $\mathcal{D} = \frac{x \in R \text{ in } \Delta}{\Delta \vdash_{\Sigma} x \in R} \text{sort_var.}$

Then we choose $S = R$ ¹⁷ and $R \leq R$ (by rule `sub_reflex`).

Case: $\mathcal{D} = \frac{\mathcal{D}_1 \quad \Delta \vdash_{\Sigma} x \in R_1 \quad R_1 \leq R}{\Delta \vdash_{\Sigma} x \in R} \text{sort_subs.}$

Then there is S such that $S \leq R_1$ and $x \in S$ in Δ (by the induction hypothesis on \mathcal{D}_1), and so $S \leq R$ (by rule `sub_trans`).

Case: $\mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Delta \vdash_{\Sigma} x \in R_1 \quad \Delta \vdash_{\Sigma} x \in R_2} \text{sort_inter.}$

Then $R = R_1 \& R_2$.

¹⁷(since x can only appear once in Δ)

Now $S \leq R_1$ and $S \leq R_2$ (induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2).
Thus $S \leq R_1 \& R_2$ (by rule `sub.inter.right`).

Case: $\mathcal{D} = \frac{}{\Delta \vdash_{\Sigma} x \in \top^A}$ `sort_top`.

Then $R = \top^A$ and there must be some $x \in S$ in Δ (by the refinement restriction¹⁸). Finally, $S \leq \top^A$ (by rule `sub_top`).¹⁹

□

Lemma 3.10.4 (Inversion for Constants)

If $c \in S$ is in Σ then $\Delta \vdash_{\Sigma} c \in R$ if and only if $S \leq R$.

Proof: Almost identical to the previous proof.

□

Lemma 3.10.5 (Inversion for λ -Abstractions)

$\Delta \vdash_{\Sigma} \lambda x:A. M_2 \in R_1 \rightarrow R_2$ if and only if $\Delta, x \in R_1 \vdash_{\Sigma} M_2 \in R_2$.

Proof:

“**If**” part: By rule `sort_lam`.

“**Only if**” part: By the Inversion Lemma for \rightarrow (Lemma 3.9.5) applied to $R_1 \rightarrow R_2 \leq R_1 \rightarrow R_2$ (which is obtained by rule `sub_reflex`).

□

Theorem 3.10.6 (Decidability of Sorting)

Given $\Gamma \vdash_{\Sigma} M : A$ and $\Delta \sqsubset \Gamma$ and $R \sqsubset A$ there is a procedure for determining whether there is a derivation of $\Delta \vdash_{\Sigma} M \in R$.²⁰

Proof: By induction on the structure of the derivation $\mathcal{D} :: \Gamma \vdash_{\Sigma} M : A$ (or equivalently, by induction on the structure of M) and the structure of R lexicographically. We have the following cases.²¹

Case: $\mathcal{D} = \frac{x:A \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x : A}$ `tp_var`.

Then $M = x$ and since $\Delta \sqsubset \Gamma$ there must also be some $S \sqsubset A$ such that $x \in S$ is in Δ .

¹⁸Lemma? Or say more?

¹⁹Reformat this proof using the “proof steps” tabular format?

²⁰Perhaps the algorithm should be explicitly represented using inference rules?

²¹Consider the cases for \top .

To determine whether there is a derivation of $\Delta \vdash_{\Sigma} M \in R$ we simply use the subsorting algorithm in Section 3.8 to check whether $S \leq R$ holds.

By the above Inversion Lemma for Variables (Lemma 3.10.3), and because x appears only once in Δ ²², this correctly determines whether $\Delta \vdash_{\Sigma} x \in R$.

$$\text{Case: } \mathcal{D} = \frac{c:a \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c : a} \text{tp_const.}$$

Similar to the previous case.

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{\Gamma \vdash_{\Sigma} M_1 : B \rightarrow A} \quad \frac{\mathcal{D}_2}{\Gamma \vdash_{\Sigma} M_2 : B}}{\Gamma \vdash_{\Sigma} M_1 M_2 : A} \text{tp_app.}$$

Then $M = M_1 M_2$.

Let Σ_B be the finite set in the proof of the Finiteness of Refinements Theorem (Theorem 3.7.3).

To determine whether $\Delta \vdash_{\Sigma} M_1 M_2 \in R$ we try each $S \in \Sigma_B$ and check whether $\Delta \vdash_{\Sigma} M_1 \in S \rightarrow R$ and $\Delta \vdash_{\Sigma} M_2 \in S$ (which we can determine, by the induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2).

This correctly determines whether $\Delta \vdash_{\Sigma} M_1 M_2 \in R$ holds because this is so iff there is a sort S such that $\Delta \vdash_{\Sigma} M_1 \in S \rightarrow R$ and $\Delta \vdash_{\Sigma} M_2 \in S$ (by the Inversion Lemma for Applications above, 3.10.2), and it suffices to consider only $S \in \Sigma_B$ (by rule `sort_sub` and the Finiteness of Refinements Theorem).

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_2}{\Gamma, x:A \vdash_{\Sigma} M' : B}}{\Gamma \vdash_{\Sigma} \lambda x:A. M' : A \rightarrow B} \text{tp_lam.}$$

Then $M = \lambda x:A. M'$. We have three subcases.

Subcase: $R = R_1 \rightarrow R_2$.

We simply determine whether $\Delta, x \in R_1 \vdash_{\Sigma} M' \in R_2$, using the procedure obtained by applying the induction hypothesis to \mathcal{D}_2, R_2 .

²²lemma?

By the Inversion Lemma for λ -Abstractions (Lemma 3.10.5) this correctly determines $\Delta \vdash \lambda x:A. M' \in R_1 \rightarrow R_2$, as required.

Subcase: $R = R_1 \& R_2$.

Then

$$\Delta \vdash \lambda x:A. M' \in R_1 \& R_2$$

iff

$$\Delta \vdash \lambda x:A. M' \in R_1 \quad \text{and} \quad \Delta \vdash \lambda x:A. M' \in R_2$$

(by rule `sort_inter` in one direction; by rules `sub_inter_left_2`, `sub_inter_left_2` and `sort_subs` in the other)²³.

We thus simply determine whether both $\Delta \vdash \lambda x:A. M' \in R_1$ and $\Delta \vdash \lambda x:A. M' \in R_2$ hold by using the procedures obtained by applying the induction hypothesis to \mathcal{D}, R_1 and \mathcal{D}, R_2 respectively.

Subcase: $R = \top^A$.

Then we determine that $\Delta \vdash_{\Sigma} \lambda x:A. M' \in \top^A$ holds, by rule `sort_top`.

24

□

3.11 Bidirectional Sort Checking

The proofs in the previous section do not lead to practical algorithms for inferring principal sorts for terms nor for deciding whether a term has a particular sort. This is because they rely on enumerating the set of unique refinements of a type, which can be huge. Previous work on refinements for ML by Freeman [Fre94] has attempted to construct an practical algorithm for inferring principal sorts by using techniques from abstract interpretation to avoid this enumeration as much as possible.²⁵ This appears to work quite well for programs which only use first-order functions, and which have only a small number of base sorts. Alas, for most real ML programs it seems that sort inference is infeasible, in part because the principal types themselves can be huge.

²³Make this a lemma so it can be reused in the correctness proof?

²⁴Add a corollary that sort inference is decidable? (By composing the previous two proofs.)

²⁵and efficiently generated the unique refinements by representing each base sort as a union of “unsplitable” components

However, it is unlikely that a programmer has such huge sorts in mind as they write their program. Type systems for programming languages generally assume that the programmer is aware of the types that they intend to be associated with their program, and it seems reasonable to extend this assumption to sorts. In this section we show that if we allow the programmer to declare a small number of these intended sorts then we can check the sort correctness of a term without enumerating refinements.

Our algorithm has some elements that are similar to the proof of Decidability of Sorting (Theorem 3.10.6). In particular, it checks λ -abstractions against a goal sort in essentially the same way. However for application terms it avoids the enumeration by a syntactic restriction that ensures that the sorts for the function can be inferred. This syntactic restriction seems to be quite reasonable in practice: for ML programs it generally only requires that the intended sorts be declared at `fun` definitions.

Our syntactic restriction is similar to that in the programming language Forsythe [Rey88, Rey96], which also includes intersections. Our restriction is somewhat simpler though: we only distinguish two classes of terms, while Forsythe allows more terms but requires an infinite number of syntactic classes. Our algorithm for sort checking is very different that used in Forsythe, in part due to the value restriction, but also because the Forsythe algorithm does not extend to arbitrary base lattices: it requires that $r_1 \& r_2 \leq s$ only if $r_1 \leq s$ or $r_2 \leq s$.²⁶

3.11.1 Syntax

The bidirectional checking algorithm uses the following two syntactic classes of terms, which correspond to restrictions of the previous class of terms. We also have a new term constructor for sort annotations that allows a list of alternative goal sorts: to infer sorts for a term with a sort annotation ($C \in R_1, \dots, R_n$) we only consider the sorts R_1, \dots, R_n . This is similar to the type annotations with alternatives in Forsythe [Rey96], except that in Forsythe the annotations are placed on variable bindings rather than on terms.

Inferable Terms	I	$::=$	$c \mid x \mid IC \mid (C \in \rho)$
Checkable Terms	C	$::=$	$I \mid \lambda x:A. C$
Sort Constraints	ρ	$::=$	$\cdot \mid \rho, R$

²⁶Elaborate here or in the related work in the introduction. Also mention Benli's thesis work and Local Type Inference.

3.11.2 Sort Checking Algorithm

The sorting judgments for inferable and checkable terms are defined for terms $M = I$ or C such that $\Gamma \vdash_{\Sigma} M : A$ and $\Delta \sqsubset \Gamma$ and $R \sqsubset A$. We require one auxiliary judgment to break a sort into its conjuncts.²⁸

$$\begin{aligned} \Delta \vdash_{\Sigma} I \overset{\gg}{\Leftarrow} R & \quad \text{Term } I \text{ has } R \text{ as an inferable sort.} \\ \Delta \vdash_{\Sigma} C \overset{\ll}{\Leftarrow} R & \quad \text{Term } C \text{ checks against sort } R. \\ R \overset{\gg}{\Leftarrow} S & \quad \text{Sort } R \text{ has } S \text{ as a conjunct.} \end{aligned}$$

It is our intention that the rules for these judgments be interpreted algorithmically as follows.

1. Given Δ, Σ and I , we can construct all R such that there is a derivation of $\Delta \vdash_{\Sigma} I \overset{\gg}{\Leftarrow} R$.
2. Given Δ, Σ, C , and R we can check whether there is a derivation of $\Delta \vdash_{\Sigma} C \overset{\ll}{\Leftarrow} R$.
3. Given R we can construct all S such that there is a derivation of $R \overset{\gg}{\Leftarrow} S$.

$$\begin{array}{c} \frac{}{R_1 \rightarrow R_2 \overset{\gg}{\Leftarrow} R_1 \rightarrow R_2} \text{subout_reflex} \\ \\ \frac{R_1 \overset{\gg}{\Leftarrow} S_1}{R_1 \& R_2 \overset{\gg}{\Leftarrow} S_1} \text{subout_inter1} \qquad \frac{R_2 \overset{\gg}{\Leftarrow} S_2}{R_1 \& R_2 \overset{\gg}{\Leftarrow} S_2} \text{subout_inter2} \end{array}$$

²⁷Include CI as checkable? This allows a β -redex to be used as a let without respecifying the sorts for the (checkable) body of the let. Not needed if we allow annotations with alternatives. With CI the formation rules are less well-behaved.

²⁸Restrict to simplified sorts here, as in the implementation?

$$\begin{array}{c}
\frac{x \in R \text{ in } \Delta}{\Delta \vdash_{\Sigma} x \gg R} \text{si_var} \qquad \frac{c \in R \text{ in } \Sigma}{\Delta \vdash_{\Sigma} c \gg R} \text{si_const} \\
\\
\frac{\Delta \vdash_{\Sigma} I \gg R \quad R \gg S_1 \rightarrow S_2 \quad \Delta \vdash_{\Sigma} C \ll S_1}{\Delta \vdash_{\Sigma} IC \gg S_2} \text{si_app} \\
\\
\frac{R \text{ in } \rho \quad \Delta \vdash_{\Sigma} C \ll R}{\Delta \vdash_{\Sigma} (C \in \rho) \gg R} \text{si_annot} \\
\\
\frac{\Delta, x \in R \vdash_{\Sigma} C \ll S}{\Delta \vdash_{\Sigma} \lambda x. C \ll R \rightarrow S} \text{sc_lam} \qquad \frac{\Delta \vdash_{\Sigma} I \gg R \quad |R| \triangleq |S|}{\Delta \vdash_{\Sigma} I \ll S} \text{sc_subs} \\
\\
\frac{\Delta \vdash_{\Sigma} \lambda x:A. C \ll R \quad \Delta \vdash_{\Sigma} \lambda x:A. C \ll S}{\Delta \vdash_{\Sigma} \lambda x:A. C \ll R \& S} \text{sc_inter} \\
\\
\frac{}{\Delta \vdash_{\Sigma} \lambda x:A_1. C \ll \top^{A_1 \rightarrow A_2}} \text{sc_top}
\end{array}$$

29

3.11.3 Soundness and Completeness

We can not directly check the soundness and completeness of this algorithm with respect to the declarative sorting rules: there is no declarative rule for an annotated term $(C \in \rho)$. Two possible solutions to this are:

1. Relate the declarative and algorithmic systems via an erasure function which removes all annotations from a checkable term, and an annotation function which produces a checkable term from a declarative sorting derivation.

²⁹A simpler option here (suggested by Frank) is to have rules *si_inter1,2*, but this looks less “algorithmic” to me. It also adds some unnecessary non-determinism at rule *sc_subs*. It also makes the purpose of the lemma for the completeness of *si_app* somewhat harder to see.

2. Extend the declarative system with a rule for annotations, and show that the algorithmic and extended declarative systems are equivalent for checkable terms.

We choose the second solution because the first would leave us without a declarative system for annotated terms. This would be somewhat unsatisfactory: when considering whether an annotated term is correct the programmer would either need to follow the sort checking algorithm exactly, or make sure that the term is annotated exactly as specified by the annotation function. The declarative system allows more flexible forms of reasoning about sort correctness, and since the programmer needs to reason about the correctness of annotated terms, it is useful to extend the declarative system to include annotations.

This allows the programmer some flexibility in exactly where they add annotations: they need only ensure that their annotated program is declaratively sort correct, and that it has enough annotations to satisfy the grammar for a checkable term. We will still be interested in an annotation function though: in this context it guarantees that there is always some way of adding annotations to a declaratively well-sorted term.

We thus extend the language of declarative terms, typing rules and the declarative sorting rules as follows.

Terms $M ::= \dots \mid (M \in \rho)$

$$\frac{\rho \sqsubset A \quad \Gamma \vdash_{\Sigma} M : A}{\Gamma \vdash_{\Sigma} (M \in \rho) : A} \text{tp_annot}$$

$$\frac{R \text{ in } \rho \quad \Delta \vdash_{\Sigma} M \in R}{\Delta \vdash_{\Sigma} (M \in \rho) \in R} \text{sort_annot}$$

The rule `tp_annot` above requires an auxiliary judgment to check that the sorts in the annotation refine the correct type.

$\rho \sqsubset A$ ρ contains refinements of type A .

$$\frac{}{\cdot \sqsubset A} \text{annot_empty} \quad \frac{\rho \sqsubset A \quad R \sqsubset A}{(\rho, R) \sqsubset A} \text{annot_nempty}$$

Annotations are only used during sort-checking, so there is no reason to extend the whole development in this chapter to terms with annotations. Instead we define the function *erase* that removes annotations from a term, reflecting our intention that sort annotations should be removed before considering terms in other contexts.

$$\begin{aligned}
\text{erase}(M \in \rho) &= \text{erase}(M) \\
\text{erase}(x) &= x \\
\text{erase}(c) &= c \\
\text{erase}(\lambda x:A. M) &= \lambda x:A. \text{erase}(M) \\
\text{erase}(M N) &= \text{erase}(M) \text{erase}(N)
\end{aligned}$$

We now demonstrate that the function *erase* preserves types and sorts, justifying the notion that annotations can be removed once sort checking has been done. We will need the following lemma.

Lemma 3.11.1 (Value Erasure)

If M is a value then $\text{erase}(M)$ is a value.

Proof: By a straightforward induction on the structure of M .³⁰ □

Lemma 3.11.2 (Typing Erasure)

If $\Gamma \vdash_{\Sigma} M : A$ then $\Gamma \vdash_{\Sigma} \text{erase}(M) : A$.

Proof: By a straightforward induction on the structure of the derivation. We show the case for the rule `tp_annot`. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

$$\text{Case: } \frac{\rho \sqsubset A \quad \Gamma \vdash_{\Sigma} N : A \quad \mathcal{E}}{\Gamma \vdash_{\Sigma} (N \in \rho) : A} \text{tp_annot.}$$

Then $\text{erase}(M) = \text{erase}(N \in \rho) = \text{erase}(N)$ and we apply the induction hypothesis to \mathcal{E} to obtain $\Gamma \vdash_{\Sigma} \text{erase}(N) : A$, as required.

□

Lemma 3.11.3 (Sorting Erasure)

If $\Delta \vdash_{\Sigma} M \in R$ then $\Delta \vdash_{\Sigma} \text{erase}(M) \in A$.

³⁰Only the case for $cV_1 \dots V_n$ requires the induction.

Proof: By a straightforward induction on the structure of the derivation. We show the cases for rules `sort_annot` and `sort_inter`. The remaining cases simply rebuild the derivation by mirroring the structure of the given derivation.

$$\text{Case: } \frac{R \text{ in } \rho \quad \Delta \vdash_{\Sigma} N \in R \quad \mathcal{D}}{\Delta \vdash_{\Sigma} (N \in \rho) \in R} \text{sort_annot.}$$

Then $\text{erase}(M) = \text{erase}(N \in \rho) = \text{erase}(N)$ and we apply the induction hypothesis to \mathcal{D} to obtain $\Delta \vdash_{\Sigma} \text{erase}(N) \in R$, as required.

$$\text{Case: } \frac{\Delta \vdash_{\Sigma} V \in R_1 \quad \Delta \vdash_{\Sigma} V \in R_2 \quad \mathcal{D}_1 \quad \mathcal{D}_2}{\Delta \vdash_{\Sigma} V \in R_1 \ \& \ R_2} \text{sort_inter.}$$

Then $\text{erase}(M) = \text{erase}(V)$. We apply the induction hypothesis to \mathcal{D}_1 and \mathcal{D}_2 to obtain $\Delta \vdash_{\Sigma} \text{erase}(V) \in R_1$ and $\Delta \vdash_{\Sigma} \text{erase}(V) \in R_1$. We then apply rule `sort_inter` to rebuild the derivation, using the above lemma to satisfy the requirement that $\text{erase}(V)$ is a value.

□

We now show that for inferable and checkable terms our algorithm is correct with respect to this extended declarative system. In Section 3.12 we will extend this result to correctness with respect to the original declarative system by relating annotated and unannotated terms in the declarative system via an annotation function.

The proof of the soundness result is relatively straightforward: each of the algorithmic sort checking rules corresponds to a derivable rule for the declarative sorting judgment.

First, we have the following lemma.

Lemma 3.11.4 (Soundness of $\overset{\gg}{\triangleleft}$) *If $R \overset{\gg}{\triangleleft} S$ then $R \leq S$.*

Proof: By induction on R . We have the following three cases, by inversion on the last step in the derivation of $R \overset{\gg}{\triangleleft} S$.

Case: The last inference step is an instance of `subout_reflex`. By rule `sub_reflex`.

$$\text{Case: } \frac{R_1 \overset{\gg}{\leq} S_1}{R_1 \& R_2 \overset{\gg}{\leq} S_1} \text{ subout_inter1.}$$

Then $R_1 \leq S_1$ (by ind. hyp.), and $R_1 \& R_2 \leq R_1$ (by `sub_inter_left_1`), thus $R_1 \& R_2 \leq S_1$ (by `sub_trans`).

$$\text{Case: } \frac{R_2 \overset{\gg}{\leq} S_2}{R_1 \& R_2 \overset{\gg}{\leq} S_2} \text{ subout_inter2.}$$

Symmetric to the previous case.

□

Theorem 3.11.5 (Soundness of Sort Checking)

1. If $\Delta \vdash_{\Sigma} I \overset{\gg}{\in} R$ then $\Delta \vdash_{\Sigma} I \in R$.
2. If $\Delta \vdash_{\Sigma} C \overset{\ll}{\in} S$ then $\Delta \vdash_{\Sigma} C \in S$.

Proof:

By simultaneous structural induction on the derivations $\mathcal{D} :: \Delta \vdash_{\Sigma} I \overset{\gg}{\in} R$ and $\mathcal{E} :: \Delta \vdash_{\Sigma} C \overset{\ll}{\in} S$.

We have the following cases for part 1.

Case: \mathcal{D} is an instance of `si_var` or `si_const`. Immediate using rule `sort_var` or `sort_const`.

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{\Delta \vdash_{\Sigma} I \overset{\gg}{\in} S} \quad S \overset{\gg}{\leq} R_2 \rightarrow R \quad \frac{\mathcal{E}_2}{\Delta \vdash_{\Sigma} C \overset{\ll}{\in} R_2}}{\Delta \vdash_{\Sigma} IC \overset{\gg}{\in} R} \text{ si_app.}$$

$\Delta \vdash_{\Sigma} I \in S$	By ind. hyp. on \mathcal{D}_1
$S \leq R_2 \rightarrow R$	By the above lemma
$\Delta \vdash_{\Sigma} I \in R_2 \rightarrow R$	By rule <code>sort_subs</code>
$\Delta \vdash_{\Sigma} C \in R_2$	By ind. hyp. on \mathcal{E}_2
$\Delta \vdash_{\Sigma} IC \in R$	By rule <code>sort_app</code>

$$\text{Case: } \mathcal{D} = \frac{\frac{\mathcal{D}_1}{\Delta \vdash_{\Sigma} I \overset{\gg}{\in} R_1 \& R_2}}{\Delta \vdash_{\Sigma} I \overset{\gg}{\in} R_1} \text{ si_inter1.}$$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} I \in R_1 \& R_2 & \text{By ind. hyp. on } \mathcal{D}_1 \\
R_1 \& R_2 \leq R_1 & \text{By rule sub_inter.left.1} \\
\Delta \vdash_{\Sigma} I \in R_1 & \text{By rule sort_subs}
\end{array}$$

Case: \mathcal{D} is an instance of `si.inter2`. Symmetric to the previous case.

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{E}_1 \\ R \text{ in } \rho \quad \Delta \vdash_{\Sigma} C \ll R \end{array}}{\Delta \vdash_{\Sigma} (C \in \rho) \gg R} \text{si_annot.}$$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} C \in R & \text{By ind. hyp. on } \mathcal{E}_1 \\
\Delta \vdash_{\Sigma} (C \in \rho) \in R & \text{By rule sort_annot}
\end{array}$$

For part 2 we have the following cases.

$$\text{Case: } \mathcal{E} = \frac{\begin{array}{c} \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} I \gg S \quad |S| \leq |R| \end{array}}{\Delta \vdash_{\Sigma} I \ll R} \text{sc_subs.}$$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} I \in S & \text{By ind. hyp. on } \mathcal{D}_1 \\
S \leq R & \text{By Theorem 3.8.6 (Correctness of } \leq \text{)} \\
\Delta \vdash_{\Sigma} I \in R & \text{By rule sort_subs}
\end{array}$$

$$\text{Case: } \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \quad \mathcal{E}_2 \\ \Delta \vdash_{\Sigma} \lambda x:A. C \ll R_1 \quad \Delta \vdash_{\Sigma} \lambda x:A. C \ll R_2 \end{array}}{\Delta \vdash_{\Sigma} \lambda x:A. M \ll R_1 \& R_2} \text{sc_inter.}$$

By the induction hypothesis on \mathcal{E}_1 and \mathcal{E}_2 followed by rule `sort.inter`.

$$\text{Case: } \mathcal{E} = \frac{}{\Delta \vdash_{\Sigma} M_1 M_2 \ll \top^A} \text{sc_top.}$$

By rule `sort.top`

$$\text{Case: } \mathcal{E} = \frac{\begin{array}{c} \mathcal{E}_1 \\ \Delta, x \in R_1 \vdash_{\Sigma} M \ll R_2 \end{array}}{\Delta \vdash_{\Sigma} \lambda x:A. M \ll R_1 \rightarrow R_2} \text{sc_lam.}$$

By the induction hypothesis on \mathcal{E}_1 followed by rule `sort.lam`.

□

The proof of completeness of sort checking is a little more difficult: the declarative sorting rules allow derivations to be structured in ways that cannot be directly mirrored in the algorithmic system. However, the strong inversion properties demonstrated in Section 3.10 allow a reasonably direct proof by induction on the structure of terms. The following lemma extends these properties to annotated terms.

Lemma 3.11.6 (Inversion for Annotations)

$\Delta \vdash_{\Sigma} (M \in \rho) \in S$ if and only if $R \leq S$ and $\Delta \vdash_{\Sigma} M \in R$ for some R in ρ .

Proof:

The “if” part is a simple consequence of rules `sort_annot` and `sort_subs`. The “only if” part is proved by structural induction on the derivation $\mathcal{D} :: \Delta \vdash_{\Sigma} (M \in \rho) \in S$. Annotated terms are not considered to be values³¹, so we have only the following two cases.

$$\text{Case: } \mathcal{D} = \frac{\mathcal{E}_1 \quad \mathcal{D}_2}{\Delta \vdash_{\Sigma} (M \in \rho) \in S} \text{sort_annot.}$$

(Note: In the original image, the numerator is split as $S \text{ in } \rho$ and $\Delta \vdash_{\Sigma} M \in S$)

We have the required result with $R = S$, since $S \leq S$ (by `sub_reflex`) and $\mathcal{E}_1, \mathcal{D}_2$ satisfy the remaining two requirements.

$$\text{Case: } \mathcal{D} = \frac{\mathcal{D}_1 \quad \mathcal{E}_2}{\Delta \vdash_{\Sigma} (M \in \rho) \in S} \text{sort_sub.}$$

(Note: In the original image, the numerator is split as $\Delta \vdash_{\Sigma} (M \in \rho) \in S'$ and $S' \leq S$)

$\Delta \vdash_{\Sigma} (M \in \rho) \in R'$ and
 $R' \leq S'$ for some R' in ρ By ind. hyp. on \mathcal{D}_1
 $R' \leq S$ By rule `sub_trans` using \mathcal{E}_2
 As required, with $R = R'$.

□

We will also require the following lemma in the case for an application in the completeness proof.

³¹ They could be when M is a value, but this would complicate this lemma and proof, and the programmer can add the desired intersections to ρ instead.

Lemma 3.11.7 (Completeness of $\overset{\gg}{\triangleleft}$)

If $R \triangleleft S_1 \rightarrow S_2$ then there are some R_1, R_2 such that $R \overset{\gg}{\triangleleft} R_1 \rightarrow R_2$ and $R_1 \rightarrow R_2 \triangleleft S_1 \rightarrow S_2$.³²

Proof: By induction on R . We have three cases for the last step of the derivation $\mathcal{D} :: R \triangleleft S_1 \rightarrow S_2$ (the other cases do not match $S_1 \rightarrow S_2$).

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ S_1 \triangleleft R'_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ R'_2 \triangleleft S_2 \end{array}}{R'_1 \rightarrow R'_2 \triangleleft S_1 \rightarrow S_2} \text{subalg_arrow.}$$

Then $R = R'_1 \rightarrow R'_2$ and $R'_1 \rightarrow R'_2 \overset{\gg}{\triangleleft} R'_1 \rightarrow R'_2$ (by rule `subout_reflex`), and we choose $R_1 = R'_1$ and $R_2 = R'_2$.

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ R'_1 \triangleleft S_1 \rightarrow S_2 \end{array}}{R'_1 \& R'_2 \triangleleft S_1 \rightarrow S_2} \text{subalg_interL1.}$$

Then $R = R'_1 \& R'_2$.

$R'_1 \overset{\gg}{\triangleleft} R''_1 \rightarrow R''_2$ for some R''_1, R''_2 s.t.

$R''_1 \rightarrow R''_2 \triangleleft S_1 \rightarrow S_2$

By ind. hyp. on R'_1

$R'_1 \& R'_2 \overset{\gg}{\triangleleft} R''_1 \rightarrow R''_2$

By `subout_inter1`

Thus the result follows with $R_1 = R''_1$ and $R_2 = R''_2$.

$$\text{Case: } \mathcal{D} = \frac{\begin{array}{c} \mathcal{D}_1 \\ R'_2 \triangleleft S_1 \rightarrow S_2 \end{array}}{R'_1 \& R'_2 \triangleleft S_1 \rightarrow S_2} \text{subalg_interL2.}$$

Symmetric to the previous case.

□

We use the following slightly generalized form for the completeness theorem to support the induction in the proof.

Theorem 3.11.8 (Completeness of Sort Checking)

1. If $\Delta \vdash_{\Sigma} I \in R$ then there is some R' such that $\Delta \vdash_{\Sigma} I \overset{\gg}{\in} R'$ and $R' \leq R$.

³²Actually, we know $S_1 \triangleleft R_1$ and $R_2 \triangleleft S_2$: maybe it would be better to make this explicit here. Later $R_1 \rightarrow R_2$ will be an “ordinary” sort (non-intersection, non-base sort).

2. If $\Delta \vdash_{\Sigma} C \in R$ and $R \leq R'$ then $\Delta \vdash_{\Sigma} C \ll R'$.

33

Proof:

By structural induction on I and C . We order Part 1 of the theorem before Part 2, i.e. in the proof of Part 2 when C is an inferable term we allow an appeal to Part 1 of the induction hypothesis for the same term, but not vice-versa.

We have the following cases for I in Part 1 of the theorem.

Case: $I = I_1 C_2$.

$\Delta \vdash_{\Sigma} I_1 C_2 \in R$	Assumption
$\Delta \vdash_{\Sigma} I_1 \in R_2 \rightarrow R$ and	
$\Delta \vdash_{\Sigma} C_2 \in R_2$	for some R_2
	By Inv. Lemma (3.10.2)
$\Delta \vdash I_1 \gg R'_1$	for some $R'_1 \leq R_2 \rightarrow R$
	By ind. hyp. (1)
$R'_1 \gg R'_2 \rightarrow R'$	for some $R'_2 \rightarrow R' \leq R_2 \rightarrow R$
	By Compl- \gg (3.11.7)
$R_2 \leq R'_2$ and $R' \leq R$	By inversion
$\Delta \vdash_{\Sigma} C_2 \ll R'_2$	By ind. hyp. (2)
$\Delta \vdash_{\Sigma} I_1 C_2 \gg R'$	By rule <code>si_app</code>

As required, with R' as above.

Case: $I = x$.

$\Delta \vdash_{\Sigma} x \in R$	Assumption
$x \in S$ in Δ	for some $S \leq R$
	By Inv. Lemma (3.10.3)
$\Delta \vdash_{\Sigma} x \gg S$	By rule <code>si_var</code>

As required, with $R' = S$.

Case: $I = c$.

Similar to the previous case.

Case: $I = (C \in \rho)$.

$\Delta \vdash_{\Sigma} (C \in \rho) \in R$	Assumption
$\Delta \vdash_{\Sigma} C \in S$ and	
$S \leq R$	for some S in ρ
	By Inv. Lemma (3.11.6)
$\Delta \vdash_{\Sigma} C \ll S$	By ind. hyp. (2)
$\Delta \vdash_{\Sigma} (C \in \rho) \gg S$	By rule <code>si_annot</code>

As required, with $R' = S$.

³³ Was: $\Delta' \leq \Delta$, but not needed here (it's taken care of by the Inversion Lemma for \rightarrow), so I removed it.

Case: $I = \lambda x:A_1. C$. Cannot occur, because Part 1 requires an inferable term.

We have the following two cases for C in Part 2 of the theorem.

Case: C is an inferable term.

$$\begin{array}{ll} \Delta \vdash_{\Sigma} C \gg \in S' & \text{for some } S' \leq R & \text{By ind. hyp. (1)} \\ S' \leq R' & & \text{By rule sub.trans} \\ \Delta \vdash_{\Sigma} C \ll \in R' & & \text{By rule sc.subs} \end{array}$$

Case: $C = \lambda x:A_1. C_2$.

$$\begin{array}{ll} \text{Then} & \\ \Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \in R & \text{Assumption} \\ R \leq R' & \text{Assumption} \end{array}$$

We now need to prove that $\Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in R'$. We do this by proving the more general result that for any S' such that $R \leq S'$ we have $\Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in S'$ (taking the instance $S' = R'$ gives the required result). We use a nested induction on S' .

We have three sub-cases for S' in this nested induction.

Subcase: $S' = S'_1 \rightarrow S'_2$.

$$\begin{array}{ll} R \leq S'_1 \rightarrow S'_2 & \text{Assumption} \\ \Delta, x \in S'_1 \vdash_{\Sigma} C_2 \in S'_2 & \text{By Inv. Lemma for } \rightarrow \text{ (3.9.5)} \\ \Delta, x \in S'_1 \vdash_{\Sigma} C_2 \ll \in S'_2 & \text{By ind. hyp. (2)} \\ \Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in S'_1 \rightarrow S'_2 & \text{By rule sc.lam} \end{array}$$

Subcase: $S' = S'_1 \& S'_2$.

$$\begin{array}{ll} R \leq S'_1 \& S'_2 & \text{Assumption} \\ R \leq S'_1 \text{ and } R \leq S'_2 & \text{By rules sub.trans, sub.left.1,2} \\ \Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in S'_1 & \text{By nested ind. hyp.} \\ \Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in S'_2 & \text{By nested ind. hyp.} \\ \Delta \vdash_{\Sigma} \lambda x:A_1. C_2 \ll \in S'_1 \& S'_2 & \text{By rule sc.inter} \end{array}$$

Subcase: $S' = \top_1^A$.

Immediate using rule sc.top.

□

³⁴ Using the Inv. Lemma avoids the need to generalise the whole theorem to $\Delta' \leq \Delta$.

3.12 Annotatability

We now demonstrate that every well-sorted unannotated term can be annotated to produce a checkable term. This is done by defining an annotation function that is a right inverse of the function *erase* for well-sorted terms.³⁵

Recall that the proof of the Sorting Erasure Lemma (3.11.3) maps a sorting derivation for an annotated term M to a sorting derivation for the unannotated term $\text{erase}(M)$. We are specifically interested in the case where M is a checkable term. To demonstrate that well-sorted terms can always be annotated appropriately we need to prove a theorem that maps in the opposite direction, i.e. a theorem roughly like the following.

If $\Delta \vdash_{\Sigma} M \in R$ then we can construct a checkable term C such that $|C| = M$ and $\Delta \vdash_{\Sigma} C \in R$ (and similarly for inferable terms).

However, a difficulty arises when we attempt to prove such a theorem using a standard structural induction on sorting derivations. The main difficulty arises when we have a sorting derivation of the following form.

$$\frac{\begin{array}{c} \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} V \in R_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} V \in R_2 \end{array}}{\Delta \vdash_{\Sigma} V \in R_1 \ \& \ R_2} \text{sort_inter}$$

Then, applying the the induction hypothesis to \mathcal{D}_1 and \mathcal{D}_2 yields two terms C_1 and C_2 such that $|C_1| = V$, $|C_2| = V$, $\Delta \vdash_{\Sigma} C_1 \in R_1$ and $\Delta \vdash_{\Sigma} C_2 \in R_2$. But then we cannot apply the rule `sort_inter` since C_1 and C_2 may be different terms.

One possible solution to this difficulty is to generalize the induction hypothesis so that it directly produces a single term from multiple sorting derivations, i.e. roughly like the following, which would require induction on the sum of the sizes of the assumed derivations.

If $\Delta_1 \vdash_{\Sigma} M \in R_1$ and $\Delta_2 \vdash_{\Sigma} M \in R_2$ and \dots and $\Delta_n \vdash_{\Sigma} M \in R_n$ then we can construct C such that $|C| = M$ and $\Delta_1 \vdash_{\Sigma} C \in R_1$ and $\Delta_2 \vdash_{\Sigma} C \in R_2$ and \dots and $\Delta_n \vdash_{\Sigma} C \in R_n$

³⁵Except that it depends on the sorting derivation.

It appears that this approach would succeed, but the use of lists would be somewhat notationally awkward.

Here we will present another solution to the above problem by showing how to construct a term C which combines the annotations from C_1 and C_2 , such that $\Delta \vdash_{\Sigma} C \in R_1$ and $\Delta \vdash_{\Sigma} C \in R_2$. We wish to avoid some complications that arise in the case where C_1 and C_2 have annotations on different subterms. We thus restrict our attention to checkable and inferable terms with annotations on exactly those subterms where they cannot be avoided. We call such terms “minimal checkable terms” and “minimal inferable terms” to capture the intuition that they only have annotations where required by the definitions of inferable and checkable terms.³⁶

Definition 3.12.1

Minimal Inferable Terms $I^* ::= c \mid x \mid I^* C^* \mid ((\lambda x:A. C^*) \in \rho)$

Minimal Checkable Terms $C^* ::= c \mid x \mid I^* C^* \mid \lambda x:A. C^*$

37

The following function $C_1^* \bowtie C_2^*$ combines the annotations from two minimal checkable terms C_1^* and C_2^* with $|C_1^*| = |C_2^*|$. It uses an auxillary function $I_1^* \bowtie I_2^*$ to combine two minimal inferable terms with $|I_1^*| = |I_2^*|$. These functions are defined inductively following the definitions of minimal inferable terms and minimal checkable terms. The constraints $|C_1^*| = |C_2^*|$ and $|I_1^*| = |I_2^*|$ guarantee that the two terms are identical other than the choice of sort annotations.

$$\begin{aligned} c \bowtie c &= c \\ x \bowtie x &= x \\ (I_1^* C_1^*) \bowtie (I_2^* C_1^*) &= (I_1^* \bowtie I_2^*)(C_1^* \bowtie C_2^*) \\ ((\lambda x:A. C_1^*) \in \rho_1) \bowtie ((\lambda x:A. C_2^*) \in \rho_2) &= ((\lambda x:A. C_1^* \bowtie C_2^*) \in \rho_1, \rho_2) \end{aligned}$$

$$\begin{aligned} c \bowtie c &= c \\ x \bowtie x &= x \\ (I_1^* C_1^*) \bowtie (I_2^* C_1^*) &= (I_1^* \bowtie I_2^*)(C_1^* \bowtie C_2^*) \\ (\lambda x:A. C_1^*) \bowtie (\lambda x:A. C_2^*) &= (\lambda x:A. C_1^* \bowtie C_2^*) \end{aligned}$$

Here the notation ρ_1, ρ_2 means the concatenation of the two lists.

³⁶An alternative would be to require annotations on every subterm, but I think it's better to work with a minimum of annotations.

³⁷Perhaps move this definition earlier to show exactly where annotations are required? If we were to allow CI as a checkable term, then this would be more complicated.

The following lemma demonstrates that these functions have the intended properties.

Lemma 3.12.2 (Annotation Combination)

1. If $|I_1^*| = |I_2^*|$ and either $\Delta \vdash_{\Sigma} I_1^* \in R$ or $\Delta \vdash_{\Sigma} I_2^* \in R$ then $\Delta \vdash_{\Sigma} (I_1^* \bowtie I_2^*) \in R$.
2. If $|C_1^*| = |C_2^*|$ and either $\Delta \vdash_{\Sigma} C_1^* \in R$ or $\Delta \vdash_{\Sigma} C_2^* \in R$ then $\Delta \vdash_{\Sigma} (C_1^* \bowtie C_2^*) \in R$.

Proof: By structural induction on the sorting derivations $\mathcal{D}_1 :: \Delta \vdash_{\Sigma} I_1^* \in R$, $\mathcal{D}_2 :: \Delta \vdash_{\Sigma} I_2^* \in R$, $\mathcal{E}_1 :: \Delta \vdash_{\Sigma} C_1^* \in R$, and $\mathcal{E}_2 :: \Delta \vdash_{\Sigma} C_2^* \in R$.

We focus on the cases for \mathcal{D}_1 and \mathcal{E}_1 since the other two are symmetric. We show two cases: the remaining cases are similar and straightforward.

$$\text{Case: } \mathcal{D}_1 = \frac{R \text{ in } \rho \quad \begin{array}{c} \mathcal{D}_{11} \\ \Delta \vdash_{\Sigma} C_{11}^* \in R \end{array}}{\Delta \vdash_{\Sigma} (C_{11}^* \in \rho_1) \in R} \text{sort_annot.}$$

$$\begin{array}{ll} (C_{11}^* \in \rho_1) \bowtie (C_{22}^* \in \rho_2) = ((C_{11}^* \bowtie C_{22}^*) \in \rho_1, \rho_2) & \text{By def. } \bowtie, \bowtie \\ \Delta \vdash_{\Sigma} (C_{11}^* \bowtie C_{22}^*) \in R & \text{By ind. hyp. on } \mathcal{D}_{11} \\ R \text{ in } \rho_1, \rho_2 & \text{By def. } \rho_1, \rho_2 \\ \Delta \vdash_{\Sigma} ((C_{11}^* \bowtie C_{22}^*) \in \rho_1, \rho_2) \in R & \text{By rule sort_annot} \end{array}$$

$$\text{Case: } \mathcal{E}_1 = \frac{\begin{array}{c} \mathcal{E}_{11} \\ \Delta \vdash_{\Sigma} C_1^* \in R_1 \end{array} \quad \begin{array}{c} \mathcal{E}_{12} \\ \Delta \vdash_{\Sigma} C_1^* \in R_2 \end{array}}{\Delta \vdash_{\Sigma} C_1^* \in R_1 \ \& \ R_2} \text{sort_inter}$$

$$\begin{array}{ll} \text{with } C_1^* \text{ a value.} & \\ C_1^* = c \text{ or } x \text{ or } \lambda x:A. C_{11}^* & \text{By def. } C^*, \text{ values} \\ C_1^* \bowtie C_2^* \text{ is a value} & \text{By def. } \bowtie, \text{ values} \\ \Delta \vdash_{\Sigma} (C_1^* \bowtie C_2^*) \in R_1 & \text{By ind. hyp. on } \mathcal{E}_1 \\ \Delta \vdash_{\Sigma} (C_1^* \bowtie C_2^*) \in R_2 & \text{By ind. hyp. on } \mathcal{E}_2 \\ \Delta \vdash_{\Sigma} (C_1^* \bowtie C_2^*) \in R_1 \ \& \ R_2 & \text{By rule sort_inter} \end{array}$$

□

Theorem 3.12.3 (Annotatability)

If $\Delta \vdash_{\Sigma} M \in R$ then we can construct a minimal inferable term I^* and a minimal checkable term C^* such that $|I^*| = M$ and $\Delta \vdash_{\Sigma} I^* \in R$ and $|C^*| = M$ and $\Delta \vdash_{\Sigma} C^* \in R$.

Proof: By induction on the sorting derivation. We show two cases. The remaining cases simply rebuild the term, using the induction hypothesis on sub-derivations.

$$\text{Case: } \frac{\begin{array}{c} \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} V \in R_1 \end{array} \quad \begin{array}{c} \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} V \in R_2 \end{array}}{\Delta \vdash_{\Sigma} V \in R_1 \ \& \ R_2} \text{sort_inter.}$$

If $V = c$ or $V = x$ then V already has the required forms, so we set $I^* = V$ and $C^* = V$.

Otherwise $V = \lambda x:A. M_1$, and then

$$\begin{array}{lll} \Delta \vdash_{\Sigma} C_1^* \in R_1 & \text{and} & \\ |C_1^*| = V & \text{for some } C_1^* & \text{By ind. hyp. on } \mathcal{D}_1 \\ \Delta \vdash_{\Sigma} C_2^* \in R_2 & \text{and} & \\ |C_2^*| = V & \text{for some } C_2^* & \text{By ind. hyp. on } \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} C^* \in R_1 & \text{and} & \\ \Delta \vdash_{\Sigma} C^* \in R_2 & \text{and} & \\ |C^*| = V & \text{for some } C^* & \text{By above lemma on } C_1^*, C_2^* \\ C^* = \lambda x:A. C_3^* & & \text{By def. } |\cdot| \\ C^* \text{ is a value} & & \text{By def. value} \\ \Delta \vdash_{\Sigma} C^* \in R_1 \ \& \ R_2 & \text{By rule sort_inter} \\ \Delta \vdash_{\Sigma} (C^* \in R_1 \ \& \ R_2) \in R_1 \ \& \ R_2 & \text{By rule sort_annot} \\ |(C^* \in R_1 \ \& \ R_2)| = |C^*| = M & & \text{By def. } |\cdot| \end{array}$$

Then C^* is as required, and we set $I^* = (C^* \in R_1 \ \& \ R_2)$.

$$\text{Case: } \frac{\begin{array}{c} \mathcal{D}_2 \\ \Delta, x \in R_1 \vdash_{\Sigma} M_2 \in R_2 \end{array}}{\Delta \vdash_{\Sigma} \lambda x:A. M_2 \in R_1 \rightarrow R_2} \text{sort_lam.}$$

$$\begin{array}{lll} \Delta, x \in R_1 \vdash_{\Sigma} C_2^* \in R_2 & \text{and} & \\ |C_2^*| = M_2 & \text{for some } C_2^* & \text{By ind. hyp. on } \mathcal{D}_2 \\ \Delta \vdash_{\Sigma} \lambda x:A. C_2^* \in R_1 \rightarrow R_2 & & \text{By rule sort_lam} \\ \Delta \vdash_{\Sigma} ((\lambda x:A. C_2^*) \in R_1 \rightarrow R_2) \in R_1 \rightarrow R_2 & & \text{By rule sort_annot} \end{array}$$

Then

$$|(\lambda x:A. C_2^*) \in R_1 \rightarrow R_2| = |\lambda x:A. C_2^*| = \lambda x:A. |C_2^*| = \lambda x:A. M_2$$

and we choose $C^* = \lambda x:A. C_2^*$ and $I^* = ((\lambda x:A. C_2^*) \in R_1 \rightarrow R_2)$.

□

Corollary 3.12.4 *If $\Delta \vdash_{\Sigma} M \in R$ then we can construct a checkable term C and an inferable term I such that $\Delta \vdash_{\Sigma} C \overset{\ll}{\in} R$ and there is some $R' \leq R$ such that $\Delta \vdash_{\Sigma} I \overset{\gg}{\in} R'$.*

Proof: By composing the previous theorem with the Completeness of Sort Checking Theorem (3.11.8). □

[*Add some examples here, and in the preceding.*]

Chapter 4

Soundness with effects

4.1 Introduction

In this chapter we demonstrate that the restrictions presented in Chapter 3 lead to a system that is sound in the presence of effects. We do this by considering a small call-by-value language ML^{ref} with a standard feature involving effects, namely ML mutable references. We place a value restriction on the introduction of intersections, omit the problematic distributivity rule from subtyping, and then show that this leads to a sound system by proving an appropriate progress and type preservation theorem. An analysis of the proof gives some insight as to why each of our restrictions is required.

We include general intersection types in ML^{ref} instead of restricting intersections $R \& S$ to the case where R and S are refinements of the same type.¹ We do this because the refinement restriction is orthogonal to soundness in the presence of effects. Thus, our soundness result is more general than required for refinement types, and would also apply e.g. to operator overloading via intersection types. Treating general intersection types also simplifies the presentation, since we do not need separate type and sort levels. The presentation in this chapter closely follows a paper co-authored with Frank Pfenning [DP00].

[*In what follows, I'll likely replace $M : A$ by $M \in R$ to emphasize the the relationship to the sort level of the previous chapter.]*

¹An alternative view is that we have a trivial level of types that assigns the single type \top to every term, with every sort refining this type.

4.2 Syntax

The syntax of ML^{ref} is relatively standard for a call-by-value language in the ML family. We include fixed-points with eager unrolling, and distinguish two kinds of variables: those bound in λ , **let** and **case** expressions which stand for values (denoted by x), and those bound in **fix** expressions which stand for arbitrary terms (denoted by u). As proposed by Leroy [Ler93], we could also easily admit a “by name” **let** expression. We use identifiers l to address cells in the store during evaluation.

We also include an example datatype **bits** for strings of bits, along with two subtypes **nat** for natural numbers (bit-strings without leading zeroes) and **pos** for positive natural numbers. We represent natural numbers as bit-strings in standard form, with the least significant bit rightmost and no leading zeroes. We view **0** and **1** as constructors written in postfix form, and ϵ stands for the empty string. For example, 6 would be represented as $\epsilon 110$. We include an ML-style **case** expression to deconstruct strings of bits.

$$\begin{array}{l}
 \text{Types } A ::= A_1 \rightarrow A_2 \mid A \text{ ref} \mid \mathbf{unit} \\
 \quad \mid \mathbf{bits} \mid \mathbf{nat} \mid \mathbf{pos} \mid A_1 \ \& \ A_2 \\
 \text{Terms } M ::= x \mid \lambda x. M \mid M_1 M_2 \\
 \quad \mid \mathbf{let } x = M_1 \mathbf{in } M_2 \\
 \quad \mid u \mid \mathbf{fix } u. M \\
 \quad \mid l \mid \mathbf{ref } M \mid ! M \mid M_1 := M_2 \mid () \\
 \quad \mid \epsilon \mid M \mathbf{0} \mid M \mathbf{1} \\
 \quad \mid \mathbf{case } M \mathbf{of } \epsilon \Rightarrow M_1 \mid x \mathbf{0} \Rightarrow M_2 \mid y \mathbf{1} \Rightarrow M_3
 \end{array}$$

As in the previous chapters, we use A, B for types and M, N for terms. It would be confusing to use the notation $[N/x]M$ for substitution in this chapter, because it conflicts with the standard notation for evaluation contexts $E[M]$ which we will use in the reduction semantics. Instead we write $\{N/x\}M$ for the result of substituting N for x in M .

We distinguish the following terms as *values*. We do not include expression variables u because during evaluation these may be replaced by non-values.

$$\text{Values } V ::= x \mid \lambda x. M \mid l \mid () \mid \epsilon \mid V \mathbf{0} \mid V \mathbf{1}$$

For the typing judgment, we need to assign types to variables and cells in contexts Γ and Δ , respectively. Moreover, during execution of a program

we need to maintain a store C .

Variable Contexts	Γ	$::=$	$\cdot \mid \Gamma, x:A \mid \Gamma, u:A$
Cell Contexts	Δ	$::=$	$\cdot \mid \Delta, l:A$
Store	C	$::=$	$\cdot \mid C, (l = V)$
Program States	P	$::=$	$C \triangleright M$

We assume that variables x, u and cells l can be declared at most once in a context or store. We omit leading \cdot 's from contexts, and write Γ, Γ' for the result of appending two variable disjoint contexts (and similarly for cell contexts and stores).

[*Notation change: perhaps Δ for variable contexts, and Ψ for cell contexts.*]

4.3 Subtyping

The subtyping judgment for this language has the form.

$$A \leq B \quad \text{Type } A \text{ is a subtype of } B.$$

Following Chapter 3, we have the standard rules for intersection types with the exception of the distributivity subtyping rule. We also have inclusions between the base types `bits`, `nat` and `pos`, which we build directly into the the subtyping judgment.² The `ref` type constructor is non-variant.

²*It might be better to instantiate the framework in the previous chapter, but I'll stick with this until I decide whether to change that framework to declare base subsorting in the signature, as in Chapter 2.*

$$\begin{array}{c}
\frac{}{A \leq A} \qquad \frac{A_1 \leq A_2 \quad A_2 \leq A_3}{A_1 \leq A_3} \\
\\
\frac{}{A_1 \ \& \ A_2 \leq A_1} \qquad \frac{}{A_1 \ \& \ A_2 \leq A_2} \\
\\
\frac{A \leq B_1 \quad A \leq B_2}{A \leq B_1 \ \& \ B_2} \\
\\
\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \multimap A_2 \leq B_1 \multimap B_2} \\
\\
\frac{}{\text{pos} \leq \text{nat}} \quad \frac{}{\text{nat} \leq \text{bits}} \quad \frac{A \leq B \quad B \leq A}{A \text{ ref} \leq B \text{ ref}}
\end{array}$$

We obtain an algorithmic version of subtyping roughly by following the subsorting algorithm in Chapter 3. We differ by not restricting the algorithm to simplified sorts, which is unnecessary for the current lattice of base types. We use the notation A^o for an *ordinary* type, namely one that is not an intersection, although it may contain embedded intersections. This is similar to the notation R^f for simplified function sorts in Chapter 3.

³

$A \sqsubseteq B$ Type A is algorithmically a subtype of type B .

³Perhaps the presentation here should more closely follow the preceding chapter, once that is more stable.

$$\begin{array}{c}
\frac{}{\text{pos} \trianglelefteq \text{pos}} \quad \frac{}{\text{pos} \trianglelefteq \text{nat}} \quad \frac{}{\text{pos} \trianglelefteq \text{bits}} \\
\\
\frac{}{\text{nat} \trianglelefteq \text{nat}} \quad \frac{}{\text{nat} \trianglelefteq \text{bits}} \quad \frac{}{\text{bits} \trianglelefteq \text{bits}} \\
\\
\frac{B_1 \trianglelefteq A_1 \quad A_2 \trianglelefteq B_2}{A_1 \multimap A_2 \trianglelefteq B_1 \multimap B_2} \quad \frac{A \trianglelefteq B \quad B \trianglelefteq A}{A \text{ ref} \trianglelefteq B \text{ ref}} \\
\\
\frac{}{\mathbf{unit} \trianglelefteq \mathbf{unit}} \\
\\
\frac{A_1 \trianglelefteq B^o}{A_1 \ \& \ A_2 \trianglelefteq B^o} \quad \frac{A_2 \trianglelefteq B^o}{A_1 \ \& \ A_2 \trianglelefteq B^o} \\
\\
\frac{A \trianglelefteq B_1 \quad A \trianglelefteq B_1}{A \trianglelefteq B_1 \ \& \ B_2}
\end{array}$$

We now prove three properties and show that algorithmic and declarative subtyping coincide. The properties and proofs are essentially the same as in Chapter 3.

Lemma 4.3.1 (Properties of Algorithmic Subtyping) *The algorithmic subtyping judgment satisfies:*

1. If $A \trianglelefteq B$ then $A \ \& \ A' \trianglelefteq B$ and $A' \ \& \ A \trianglelefteq B$.
2. $A \trianglelefteq A$.
3. If $A_1 \trianglelefteq A_2$ and $A_2 \trianglelefteq A_3$ then $A_1 \trianglelefteq A_3$.

Proof: By simple inductions on given types or derivations. □

Theorem 4.3.2 $A \trianglelefteq B$ if and only if $A \leq B$.

Proof: In each direction, by induction on the given derivation, using the properties in the preceding lemma. □

4.4 Typing of Terms

The typing judgment for terms has the form:

$$\Delta; \Gamma \vdash M : A \quad \text{Term } M \text{ has type } A \text{ in cell context } \Delta \\ \text{and variable context } \Gamma.$$

The typing rules are given in Figure 4.1.

These rules are standard for functions, let definitions, fixed points, references, and intersection types, with the exception that the introduction rule for intersections is restricted to values. There are three typing rules for **case**, depending on whether the subject can be shown to have type **bits**, **nat**, or **pos**. Note that the branch for ϵ does not need to be checked when the case subject has type **pos**.

The structural properties of weakening, exchange and contraction from Chapter 3 extend as expected to both cell contexts Δ and to the two kinds of variables x and u in variable contexts Γ .

Lemma 4.4.1 (Weakening, Exchange, Contraction)

1. (a) If $\Delta; \Gamma \vdash M \in R$ then
 $\Delta; (\Gamma, x \in S) \vdash M \in R$ and $\Delta; (\Gamma, u \in S) \vdash M \in R$.
- (b) If $\Delta; (\Gamma, x \in S_1, y \in S_2, \Gamma') \vdash M \in R$
then $\Delta; (\Gamma, y \in S_2, x \in S_1, \Gamma') \vdash M \in R$.
- (c) If $\Delta; (\Gamma, u_1 \in S_1, u_2 \in S_2, \Gamma') \vdash M \in R$
then $\Delta; (\Gamma, u_2 \in S_2, u_1 \in S_1, \Gamma') \vdash M \in R$.
- (d) $\Delta; (\Gamma, x \in S_1, u \in S_2, \Gamma') \vdash M \in R$
if and only if $\Delta; (\Gamma, u \in S_2, x \in S_1, \Gamma') \vdash M \in R$.
- (e) If $\Delta; (\Gamma, x \in S, y \in S, \Gamma) \vdash M \in R$
then $\Delta; (\Gamma, w \in S, \Gamma') \vdash \{w/x\}\{w/y\}M \in R$.
- (f) If $\Delta; (\Gamma, u_1 \in S, u_2 \in S, \Gamma) \vdash M \in R$
then $\Delta; (\Gamma, u_3 \in S, \Gamma') \vdash \{u_3/u_1\}\{u_3/u_2\}M \in R$.
2. (a) If $\Delta; \Gamma \vdash M \in R$ then $(\Delta, x \in S); \Gamma \vdash M \in R$.
- (b) If $(\Delta, x \in S_1, y \in S_2, \Delta'); \Gamma \vdash M \in R$ then $(\Delta, y \in S_2, x \in S_1, \Delta'); \Gamma \vdash M \in R$.
- (c) If $(\Delta, x \in S, y \in S, \Delta'); \Gamma \vdash M \in R$ then $(\Delta, w \in S, \Delta'); \Gamma \vdash \{w/x\}\{w/y\}M \in R$.

$$\begin{array}{c}
\frac{x:A \text{ in } \Gamma}{\Delta; \Gamma \vdash x : A} \text{tp_var} \quad \frac{\Delta; (\Gamma, x:A) \vdash M : B}{\Delta; \Gamma \vdash \lambda x. M : A \rightarrow B} \text{tp_lam} \quad \frac{\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : B} \text{tp_app} \\
\\
\frac{\Delta; \Gamma \vdash M : A \quad \Delta; (\Gamma, x:A) \vdash N : B}{\Delta; \Gamma \vdash \text{let } x = M \text{ in } N : B} \text{tp_let} \quad \frac{u:A \text{ in } \Gamma}{\Delta; \Gamma \vdash u:A} \text{tp_var}' \quad \frac{\Delta; (\Gamma, u:A) \vdash M : A}{\Delta; \Gamma \vdash \text{fix } u. M : A} \text{tp_fix} \\
\\
\frac{l:A \text{ in } \Delta}{\Delta; \Gamma \vdash l : A \text{ref}} \text{tp_cell} \quad \frac{\Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \text{ref } M : A \text{ref}} \text{tp_ref} \quad \frac{\Delta; \Gamma \vdash M : A \text{ref}}{\Delta; \Gamma \vdash !M : A} \text{tp_get} \\
\\
\frac{\Delta; \Gamma \vdash M : A \text{ref} \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash M := N : \text{unit}} \text{tp_set} \quad \frac{}{\Delta; \Gamma \vdash () : \text{unit}} \text{tp_unit} \\
\\
\frac{}{\Delta; \Gamma \vdash \epsilon : \text{nat}} \text{tp_e} \\
\\
\frac{\Delta; \Gamma \vdash M : \text{pos}}{\Delta; \Gamma \vdash M 0 : \text{pos}} \text{tp_z1} \quad \frac{\Delta; \Gamma \vdash M : \text{bits}}{\Delta; \Gamma \vdash M 0 : \text{bits}} \text{tp_z2} \\
\\
\frac{\Delta; \Gamma \vdash M : \text{nat}}{\Delta; \Gamma \vdash M 1 : \text{pos}} \text{tp_o1} \quad \frac{\Delta; \Gamma \vdash M : \text{bits}}{\Delta; \Gamma \vdash M 1 : \text{bits}} \text{tp_o2} \\
\\
\frac{\Delta; \Gamma \vdash M : \text{bits} \quad \Delta; \Gamma \vdash M_1 : A \quad \Delta; (\Gamma, x:\text{bits}) \vdash M_2 : A \quad \Delta; (\Gamma, y:\text{bits}) \vdash M_3 : A}{\text{case } M \text{ of } \epsilon \Rightarrow M_1 \mid x 0 \Rightarrow N_2 \mid y 1 \Rightarrow M_3 : A} \text{tp_case1} \\
\\
\frac{\Delta; \Gamma \vdash M : \text{nat} \quad \Delta; \Gamma \vdash M_1 : A \quad \Delta; (\Gamma, x:\text{pos}) \vdash M_2 : A \quad \Delta; (\Gamma, y:\text{nat}) \vdash M_3 : A}{\text{case } M \text{ of } \epsilon \Rightarrow M_1 \mid x 0 \Rightarrow N_2 \mid y 1 \Rightarrow M_3 : A} \text{tp_case2} \\
\\
\frac{\Delta; \Gamma \vdash M : \text{pos} \quad \Delta; (\Gamma, x:\text{pos}) \vdash M_2 : A \quad \Delta; (\Gamma, y:\text{nat}) \vdash M_3 : A}{\text{case } M \text{ of } \epsilon \Rightarrow M_1 \mid x 0 \Rightarrow N_2 \mid y 1 \Rightarrow M_3 : A} \text{tp_case3} \\
\\
\frac{\Delta; \Gamma \vdash V : A \quad \Delta; \Gamma \vdash V : B}{\Delta; \Gamma \vdash V : A \ \& \ B} \text{tp_conj} \quad \frac{\Delta; \Gamma \vdash M : A \quad A \sqsubseteq B}{\Delta; \Gamma \vdash M : B} \text{tp_subs}
\end{array}$$

Figure 4.1: Typing Rules

Proof: By straightforward inductions over the structure of the derivations (as in Chapter 3). \square

The value preservation lemma extends as expected. We introduce a second part to the lemma for expression variables u .

Lemma 4.4.2 (Value Preservation)

1. $\{V'/x\}V$ is a value.
2. $\{M/u\}V$ is a value.

Proof: By straightforward inductions on V . \square

The substitution lemma extends as expected to value variables x .

Lemma 4.4.3 (Value Substitution Lemma)

If $\Delta; \Gamma \vdash V : A$ and $\Delta; (\Gamma, x:A) \vdash N : B$
then $\Delta; \Gamma \vdash \{V/x\}N : B$.

Proof: By a straightforward induction on the typing derivation for N (as in Chapter 3). \square

We have an additional substitution lemma for expression variables u .

Lemma 4.4.4 (Expression Substitution Lemma)

If $\Delta; \Gamma \vdash M : A$ and $\Delta; (\Gamma, u:A) \vdash N : B$
then $\Delta; \Gamma \vdash \{M/u\}N : B$.

Proof: By induction on the typing derivation \mathcal{D}_2 for N . We show one interesting case. The remaining cases are straightforward and follow the previous substitution lemma.

$$\text{Case: } \mathcal{D}_2 = \frac{\frac{\mathcal{D}_{21}}{\Delta; (\Gamma, u \in R) \vdash V \in S_1} \quad \frac{\mathcal{D}_{22}}{\Delta; (\Gamma, u \in R) \vdash V \in S_2}}{\Delta; (\Gamma, u \in R) \vdash V \in S_1 \ \& \ S_2} \text{tp.conj.}$$

Applying the induction hypothesis to \mathcal{D}_{21} and \mathcal{D}_{22} yields the derivations $\mathcal{D}_{31} :: \Delta \vdash_{\Sigma} \{M/u\}V \in S_1$ and $\mathcal{D}_{32} :: \Delta \vdash_{\Sigma} \{M/u\}V \in S_2$. Since $\{M/u\}V$ is a value (by the second part of the value preservation lemma above) we can apply rule `sort_inter` to these derivations to obtain $\mathcal{D}_3 :: \Delta \vdash_{\Sigma} \{M/u\}V \in S_1 \ \& \ S_2$, as required.

\square

4.5 Typing of Stores and States

Stores are typed using the following judgment.

$\Delta \vdash C : \Delta'$ Store C satisfies cell context Δ'
when checked against cell context Δ .

The rules for this judgment simply require each value to have the appropriate type under the empty variable context.

$$\frac{}{\Delta \vdash \cdot : \cdot} \quad \frac{\Delta \vdash C' : \Delta' \quad \Delta; \cdot \vdash V : A}{\Delta \vdash (C', l = V) : (\Delta', l:A)}$$

The following judgment defines typing of program states.

$\vdash (C \triangleright M) : (\Delta \triangleright A)$ Program state $(C \triangleright M)$ types with
cell context Δ and type A .

It is defined directly from the previous typing judgments. We require the store C to satisfy the cell context Δ under the same cell context. This allows the consistent occurrence of cells l in the values in a cell context.

$$\frac{\Delta \vdash C : \Delta \quad \Delta; \cdot \vdash M : A}{\vdash (C \triangleright M) : (\Delta \triangleright A)}$$

4.6 Reduction Semantics

We now present a reduction style semantics for our language, roughly following Wright and Felleisen [WF94]. We start by defining *evaluation contexts*, namely expressions with a hole $[]$ within which a reduction may occur:

$$\begin{aligned} E ::= & [] \mid E M \mid V E \\ & \mid \mathbf{let} \ x = E \ \mathbf{in} \ M \\ & \mid \mathbf{ref} \ E \mid !E \mid E := M \mid V := E \\ & \mid E \ 0 \mid E \ 1 \\ & \mid \mathbf{case} \ E \ \mathbf{of} \ \epsilon \Rightarrow M_1 \mid x \ 0 \Rightarrow M_2 \mid y \ 1 \Rightarrow M_3 \end{aligned}$$

We write $C \triangleright M \mapsto C' \triangleright M'$ for a one-step computation, defined by the reduction rules in Figure 4.2. Each rule reduces a redex N that appears

$$\begin{array}{l}
C \triangleright E[(\lambda x. M) V] \mapsto C \triangleright E[\{V/x\}M] \\
C \triangleright E[\mathbf{let} x = V \mathbf{in} M] \mapsto C \triangleright E[\{V/x\}M] \\
C \triangleright E[\mathbf{fix} u. M] \mapsto C \triangleright E[\{\mathbf{fix} u. M/u\}M] \\
C \triangleright E[(\mathbf{ref} V)] \mapsto C, (l = V) \triangleright E[l] \\
\hspace{15em} (l \text{ not in } C, E) \\
C_1, (l = V), C_2 \triangleright E[!l] \mapsto C_1, (l = V), C_2 \triangleright E[V] \\
C_1, (l = V_1), C_2 \triangleright E[l := V_2] \mapsto C_1, (l = V_2), C_2 \triangleright E[()] \\
C \triangleright E[\mathbf{case} \epsilon \mathbf{of} \epsilon \Rightarrow M_1 \mid x \ 0 \Rightarrow M_2 \mid y \ 1 \Rightarrow M_3] \mapsto C \triangleright E[M_1] \\
C \triangleright E[\mathbf{case} V \ 0 \mathbf{of} \epsilon \Rightarrow M_1 \mid x \ 0 \Rightarrow M_2 \mid y \ 1 \Rightarrow M_3] \mapsto C \triangleright E[\{V/x\}M_2] \\
C \triangleright E[\mathbf{case} V \ 1 \mathbf{of} \epsilon \Rightarrow M_1 \mid x \ 0 \Rightarrow M_2 \mid y \ 1 \Rightarrow M_3] \mapsto C \triangleright E[\{V/y\}M_3]
\end{array}$$

Figure 4.2: Reduction Rules

in an evaluation position in the term M , i.e. $M = E[N]$ for some E . We maintain the invariant that M does not contain free variables x or u and that all cells l in M are defined in C .

Critical in the proof of progress are the following inversion properties, similar to Lemma 3.9.5 (Inversion for \rightarrow) which was needed in the proof of subject reduction in Chapter 3. These properties are generalizations of simpler properties in languages without subtyping, intersections, or effects.

Lemma 4.6.1 (Value Inversion)

1. If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq B_1 \multimap B_2$ then $V = \lambda x. M$ and $\Delta; (x:B_1) \vdash M : B_2$.
2. If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq B \mathbf{ref}$ then $V = l$ and there exists a B' such that $l:B'$ in Δ , $B' \trianglelefteq B$, and $B \trianglelefteq B'$.
3. If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq \mathbf{bits}$ then we have one of the following cases:
 - (a) $V = \epsilon$
 - (b) $V = (V_0 \ 0)$ and $\Delta; \cdot \vdash V_0 : \mathbf{bits}$
 - (c) $V = (V_1 \ 1)$ and $\Delta; \cdot \vdash V_1 : \mathbf{bits}$
4. If $\Delta; \cdot \vdash V : A$ and $A \trianglelefteq \mathbf{nat}$ then we have one of the following cases:
 - (a) $V = \epsilon$
 - (b) $V = (V_0 \ 0)$ and $\Delta; \cdot \vdash V_0 : \mathbf{pos}$
 - (c) $V = (V_1 \ 1)$ and $\Delta; \cdot \vdash V_1 : \mathbf{nat}$

5. If $\Delta; \cdot \vdash V : A$ and $A \sqsubseteq \text{pos}$ then we have one of the following cases:

- (a) $V = (V_0 \ 0)$ and $\Delta; \cdot \vdash V_0 : \text{pos}$
- (b) $V = (V_1 \ 1)$ and $\Delta; \cdot \vdash V_1 : \text{nat}$

Proof:

Each property is stated at a level of generality that allows it to be proved directly by inducting on the given typing derivation. For each we have inductive cases when the typing rule is subsumption or intersection introduction. The remaining cases are the introduction rules for the corresponding type constructors, and are straightforward.

We make use of the equivalence between declarative and algorithmic subtyping to reduce the number of cases that we need to consider.

[*Fill in some details here.*]

□

We are now ready to prove our main theorem, namely that our type system with mutable references and value-restricted intersections satisfies progress and type preservation, i.e., that programs can't go wrong.

[*There needs to be an example of a program going wrong somewhere in this chapter.*]

Theorem 4.6.2 (Progress and Type Preservation)

If $\vdash (C \triangleright M) : (\Delta \triangleright A)$ then either

1. M is a value.
2. $(C \triangleright M) \mapsto (C' \triangleright M')$ for some C' , M' and Δ' satisfying $\vdash (C' \triangleright M') : (\Delta, \Delta' \triangleright A)$.

Proof: By induction on the typing derivation for M .

- The case for subsumption is immediate, using the induction hypothesis.
- The case for intersection introduction is trivial: the value restriction forces M to be a value.
- For the remaining cases the typing rule matches the top term constructor of M .

- The cases for the typing rules corresponding to $\lambda x. M$, l , $()$ and ϵ are trivial, since they are values.
- The case for the typing rule corresponding to **fix** is easy, using the Expression Substitution Lemma (Lemma 4.4.4) to construct the required typing derivation.
- In the other cases, we apply the induction hypothesis to the subderivations for appropriate subterms N_i of M which are in evaluation positions i.e. $M = E[N_i]$ (in each case, there is at least one).
- Then, if for some N_i the induction hypothesis yields $(C \triangleright N_i) \mapsto (C' \triangleright N'_i)$ with $(C' \triangleright N'_i) : (\Delta, \Delta' \triangleright B)$ then we can construct the required reduction and typing derivation for M .
- Otherwise, each immediate subterm N_i with $M = E[N_i]$ is a value. For these cases we apply the appropriate clause of the preceding inversion lemma, using reflexivity of algorithmic subtyping. In each case we find that M can be reduced to some M' and that we can construct the required typing for M' , using the substitution lemma in some cases.

□

All of our restrictions are needed in this proof:

- The case of $E[!l]$ requires subtyping for A **ref** to be co-variant.
- The case of $E[l := V]$ requires subtyping for A **ref** to be contra-variant. With the previous point it means it must be non-variant.
- The value restriction is needed because otherwise the induction hypothesis is applied to the premises of the intersection introduction rule

$$\frac{\Delta; \cdot \vdash M : A_1 \quad \Delta; \cdot \vdash M : A_2}{\Delta; \cdot \vdash M : A_1 \ \& \ A_2}$$

which yields that for some C_1 , M_1 and Δ_1

$$(C \triangleright M) \mapsto (C_1 \triangleright M_1) \text{ and } \vdash (C_1 \triangleright M_1) : (\Delta, \Delta_1 \triangleright A_1)$$

and also that for some C_2 , M_2 and Δ_2

$$(C \triangleright M) \mapsto (C_2 \triangleright M_2) \text{ and } \vdash (C_2 \triangleright M_2) : (\Delta, \Delta_2 \triangleright A_2)$$

Even if we show that evaluation is deterministic (which shows $M_1 = M_2 = M'$ and $C_1 = C_2 = C'$), we have no way to reconcile Δ_1 and Δ_2 to a Δ' such that

$$\vdash (C' \triangleright M') : (\Delta, \Delta' \triangleright A_1 \ \& \ A_2)$$

because a new cell allocated in C_1 and C_2 may be assigned a different type in Δ_1 and Δ_2 . It is precisely this observation which gives rise to the following counterexample.

[*Show the counterexample here, or earlier and refer to it here.*]

- The absence of distributivity is critical in the inversion property for values $V : A$ for $A \trianglelefteq B_1 \rightarrow B_2$ which relies on the property that if $A_1 \ \& \ A_2 \trianglelefteq B_1 \rightarrow B_2$ then either $A_1 \trianglelefteq B_1 \rightarrow B_2$ or $A_2 \trianglelefteq B_1 \rightarrow B_2$.

The analysis above indicates that if we fix the cells in the store and disallow new allocations by removing the **ref** M construct, the language would be sound even without a value restriction as long as the **ref** type constructor is non-variant.

Overall, this proof is not much more difficult than the case without intersection types, but this is partially because we have set up our definitions very carefully.

4.7 Bidirectional Checking

[*Show how bidirectional checking extends to this case - general intersections, references, fix, etc.*]

4.8 Parametric Polymorphism

[*Follow the section in the ICFP paper, to show that the same ideas apply to type assignment with parametric polymorphism.*]

4.9 Examples

[*Include the examples from the ICFP paper here.*]

Chapter 5

Datatype Declarations and Pattern Matching

[*This chapter doesn't have much in it yet. I'm planning to have a lot of discussion and examples in this chapter to explain the many issues related to exactly how the inversion principles and subsorting are determined. I'll only formally treat inductive sorts - with no function or reference sorts at all. I'll include informal discussion on the extension in the implementation to all SML datatypes (other than those involving "polymorphic recursion"). Recent work on subtyping of recursive types suggests that this extension is likely to be correct. I'll also formally present sequentially pattern matching (at least).*]

The development in Chapter 3 allowed arbitrary finite lattices of refinements for each type. In this chapter I will focus on a particular mechanism for defining such lattices of refinements. This mechanism is based on the `rectype` declarations of Freeman [Fre94] which allow refinements of ML datatypes to be defined using a set of mutually recursive definitions.

The following example illustrates our mechanism for defining refinements.

Example 5.0.1 *Suppose we have a Standard ML program which contains the following datatype declaration.*

```
datatype Bits = empty | zero of Bits | one of Bits
```

Further, suppose that the program mostly uses this type to represent binary natural numbers, with the outermost `zero` or `one` representing the least

significant digit, and with no leading zeros, i.e. the most significant digit must be one.

To specify that particular parts of our program only manipulate this subset of the type `Bits`, we can define a lattice of three refinements of the type `bits` using the following `datasort` declarations.

```
datasort top_bits = empty | zero of top_bits | one of top_bits
datasort nat = empty | one of nat | zero of pos
datasort pos = one of nat | zero of pos
```

These `datasort` declarations closely follow the syntax of ordinary ML datatype declarations. However, rather than introducing a new type, they define refinements of the existing type `Bits`.

This mechanism is certainly not the only interesting way to define refinements: e.g. we might consider refinements corresponding to subsets of integer types, or refinements which capture the presence of absence of effects when an expression is executed. It seems reasonable to focus on recursive refinements for the following reasons.

- ML datatypes are closely tied to control flow via case expressions.
- These refinements include finite lattices where each element corresponds to an infinite set of values, and thus they allow case analysis to be done beyond what could be achieved by simply enumerating all possible values.
- There are many examples where these refinements would be useful when writing real programs.

The main technical differences from Freeman in this chapter are:

- I give an improved definition of subsorting that corresponds exactly to inclusion of regular-tree sets for declarations that don't include function sorts.
- I don't consider "splitting" because it results in a combinatorial explosion during sort checking. Splitting is central to the approach taken by Freeman, so this is a major departure. Instead I carefully define an *inversion principle* for each defined sort. This principle is used when sort checking a "case" expression.
- I treat general, sequential pattern matching while Freeman only considered a language with a basic elimination construct for datatypes.

[*I guess we want to use tuples here. Perhaps they should be included in chapter 4.*]

5.1 Sorts for Simple Case Expressions without Subsorting

[*I was planning to formally motivate the subsorting rules via the definable “identity coercions” in a language without subsorting. This seems to work particularly well for recursive types, but now I think that I’ll save this idea until after my thesis work.*]

- Present the typing and refinement rules. (They are the same as when we have s
- Present an operational semantics. (Or reduction rules?)
- Present the sorting rules.
- Prove sort preservation and progress here?
- Define coercions as terms equivalent to the identity (but with more sorts)

5.2 Motivating Examples

[*Examples to show that there are some choices to make when choosing inversion principles, sorts of constructors, and the cases to consider in pattern matching. I’ll need quite a few to explain all the design decisions - so far there’s just a couple here.*]

```
datatype a123 = C1 | C2 | C3
(*[ datasort a1 = C1 and a2 = C2 ]*)

datatype b = D of a123

(*[ val f : a1 -> unit & a2 -> unit ]*)
fun f C1 = ()
  | f C2 = ()

(*[ datasort a12 = C1 | C2
      datasort b12 = D of a12 ]*)

(*[ val g : b12 -> unit ]*)
fun g (D x) = f x (* Should this sort check? *)
                (* Yes: the inversion principle is
```

```

                                b12 = D of a1 | D of a2      *)

(*[ datasort b12' = D of a1 | D of a2 ]*)

(*[ val g' : b12' -> unit ]*)
fun g' (D x) = f x    (* Should this sort check? *)

(* ----- *)

datatype nat = Z | S of nat
datasort rz = Z
      and rnz = S of nat

(*[ f :> rz -> unit & rnz -> unit ]*)
fun f Z = ()
  | f (S y) = ()

(*[ g :> nat * nat -> unit ]*)
fun g (S (S y), z) => ()
  | g (y1, y2) => f y2    (* Should this sort check? *)
                        (* I think not. *)

(* Expanding to single level patterns leads to inconsistent results.
   This expansion wouldn't sort check. *)
fun g' x =
  case (fst x)
  of Z => f (snd x)
  | S x1 => (case x1
             of Z => f (snd x)
             | S y => (case (snd x) of Z => ()
                       | S _ => f (snd x)))

(* However, this expansion would sort check. *)
fun g'' x =
  case (snd x)
  of Z => (case (fst x)
             of Z => f Z
             | S x1 => (case x1 of Z => f Z
                       | S _ = () ) )
  | S x1 => f (S x1)

```

5.3 CHAPTER OUTLINE

[*What follows is an earlier plan for this chapter, but I'm considering some major changes.*]

1. Regular Sort Definitions

- (a) Datasort declaration syntax
Pure regular trees
- (b) Semantics as inductively defined subsets
- (c) Algorithms for lattice calculation
Theorem: Soundness and Completeness
- (d) Determining inversion principles
Theorem: Soundness and Completeness w.r.t. semantics
- (e) Consistency Theorems

2. A Language with Simple Deconstructors

- (a) Syntax
Effectful functions, recursion, base types, base sorts, products, unit, value constructor application and single level case.
- (b) Base judgments
 - i. Typing of Constructors
 - ii. Sorting of Constructors
 - iii. Inversion Principles
 - iv. Consistency Requirements
- (c) Typing
- (d) Operational Semantics
Theorem: Progress
- (e) Refinement relation
- (f) Declarative Subsorting
No Distributivity for products
Theorem: Finiteness of Lattices
Bounds on lattice sizes
- (g) Algorithmic Subsorting
- (h) Declarative sorting
Theorem: Subject Reduction (sorts)

- (i) Bidirectional Sort Checking
 - i. Syntax
 - Annotations, inferable and checkable terms
 - ii. Sort checking algorithm
 - iii. Soundness
 - iv. Completeness
3. A Language with Sequential Pattern Matching
- (a) Syntax
 - Effectful functions, base types, base sorts, products, unit, value constructor application and case with patterns containing constructors and products.
 - (b) Typing
 - (c) Operational Semantics
 - Theorem: Progress or Unmatched Case (types)
 - (d) Syntax for Residual Sorts
 - Including a form of union and constructor “sorts”
 - (e) Declarative Subsorting of Residual Sorts
 - (f) Subtraction of Residual Sorts
 - (g) **Declarative Sorting**
 - Use inversion principles in the rule for case (I haven’t yet managed to figure out how to formulate this in a nice, non-algorithmic way)
 - Theorem: Progress
 - (h) **Sort Checking**
 - Theorems: Soundness and Completeness

Chapter 6

Extending to full SML

[

1. *Modules*
2. *Parameterized datasorts (mostly refer to Skalka).*
3. *Variance annotations for datasort constructor parameters.*
4. *Promotion of sorts to types*
5. *Local datasort definitions*
6. *Assumptions*

]

Chapter 7

Implementation

[

1. *Library for backtracking computations with error messages.*
2. *Memoization.*
3. *Optimizations for subsort checking and lattice creation.*
4. *...*

]

Chapter 8

Experiments

[

1. *Red-Black Trees*
2. *Lambda-calculus*
3. *Elf Parser*
4. *SML/Kit Infix Resolution*

]

Chapter 9

Conclusion

Bibliography

- [AM91] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.
- [AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.
- [AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, June 1993.
- [AWL94a] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, OR, January 1994.
- [AWL94b] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages*, pages 163–173, 1994.
- [CD78] Mario Coppo and Mariangiola Dezani. A new type assignment for lambda terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 19:139–156, 1978.
- [CDCV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

- [CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.
- [DG94] Razvan Diaconescu and Joseph Goguen. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4:363–392, 1994.
- [DP97] Rowan Davies and Frank Pfenning. Practical refinement-type checking. Unpublished draft available from <http://www.cs.cmu.edu/~rowan>, July 1997.
- [DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montreal, Canada, September 2000. ACM Press.
- [DZ92] Philip W. Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 5, pages 157–187. MIT Press, Cambridge, Massachusetts, 1992.
- [FFK⁺96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.
- [FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.
- [Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.
- [Hei94] Nevin Heintze. Set based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

- [Ler93] Xavier Leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.
- [Mis84] Prakeesh Mishra. Towards a theory of types in prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984.
- [Mit84] John Mitchell. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.
- [Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.
- [Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
- [MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993. University of Nijmegen.
- [Pfe01] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2001. In preparation. Draft from April 1997 available electronically.
- [Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1991.
- [Pie93] Benjamin C. Pierce. Intersection types and bounded polymorphism. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93*, pages 346–360, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664. A version will also appear in the journal MSCS.

- [Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [Rey69] John C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68:456–461, 1969.
- [Rey81] John C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.
- [Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.
- [Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 675–700, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.
- [Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.
- [Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, June 1990.
- [Ska97] Chris Skalka. Some decision problems for ML refinement types. Master’s thesis, Carnegie-Mellon University, July 1997. To appear.
- [TDMW97] Franklyn Turbak, Allyn Dimock, Robert Muller, and Joe Wells. Compiling with polymorphic and polyvariant flow types. In *Proceedings of the First International Workshop on Types in Compilation*, June 1997. A version will appear in the *Journal of Functional Programming*.
- [WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994. Preliminary version is Rice Technical Report TR91-160.

- [Wri95] Andrew K. Wright. Simple imperative polymorphism. *Information and Computation*, 8:343–55, 1995.
- [XP97a] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. Unpublished draft available from <http://www.cs.cmu.edu/~fp>, November 1997.
- [XP97b] Hongwei Xi and Frank Pfenning. A schema for adding dependent types to ML. Unpublished draft available from <http://www.cs.cmu.edu/~fp>, July 1997.