WORKING DRAFT: Refinement Types - Dissertation Outline

Rowan Davies

February 26, 2001

1. Introduction [Much taken from proposal and updated.]
   (a) Statement of Thesis
   (b) Motivation
   (c) Overview
   (d) Related Work [Need to add quite a few here.]
   Proposal + Denney, Xi, ...
2. Intersection types and a Simply-Sorted Lambda-Calculus
   This chapter considers a lambda-calculus with intersection types and base types in a subtyping relation, algorithms for subtyping and type checking, and then introduces a simply sorted lambda-calculus with base types and lattices of refinements for each base type.

   Subtyping is natural and useful in many languages. Once we have subtyping, we often find that we would like to ascribe more than one type to a term. This is particularly apparent for functions where assigning a more accurate type to the parameter may lead to a more accurate type for the result [give an example here]. Thus, once we have subtyping, intersection types are natural and useful. This leads to an abstract view of intersections, in that our only interpretation of them is that they allow multiple types to be assigned to a term. This is similar to an abstract view of subtyping, which isn't commited to either a coercion or inclusion interpretation.

   (a) A lambda-calculus with intersection types and base types
      i. Syntax
      Use R here for types, rather than A to make the transition to refinements smoother.
      ii. Base judgments
      base subtyping, base intersection.
      Consistency requirements such as transitivity of base subtyping, and the relationship between base subtyping and base intersection.
      iii. Declarative Subtyping
      As usual for intersection types, with rules for base types.
      iv. Algorithmic Subtyping

- Mention Forsythe and Benli's algorithms, but point out the problem with subtyping between intersections of base types which makes these algorithms inappropriate for the kinds of lattices we're interested in. E.g. checking
  ```
  (A -> C12) & (A -> C23) < (A -> C2)
  ```
  fails when it should succeed if we have `C2 = (C12 & C23)`
- Judgments and inference rules for the algorithm (as in my proposal, similar to Freeman?)
  Theorem: correctness w.r.t. declarative rules

v. Declarative typing.
Theorem: substitution lemma.

vi. Reduction
Theorem: subject reduction (using algorithmic subtyping, inversion lemmas)

vii. Adding Union types (declarative only) (?)

Some of the situations where intersection types are natural also have natural subtyping relationships between unions of base types which are not captured if we only have intersections and subtyping. E.g. [give a datasort example] Union types allow us to properly model such unions. (I might include this section if I use unions later, e.g. in the declarative formulation of pattern matching, and even then perhaps it should be left until later.)

(b) Bidirectional type checking

i. Syntax
Annotations, inferable and checkable terms (compare to atoms and normal forms).

ii. Type checking algorithm

iii. Soundness

iv. Completeness w.r.t. declarative system extended with a rule for annotations.

(c) A simply-sorted lambda-calculus

In many cases we would like to rule out intersections between types that are in some sense incompatible. For example, in many situations it simply does not make sense to have an expression which has both a function type and a base type. In such situations we may want to rule out intersections and subtyping between incompatible types. One way to achieve this is to start with a typed language, and then add a "refinement type" system which allows a number of "refinements" of the existing base types, and then allow intersections of refinements which refine the same type. In this section we present a refinement-type system for the simply-typed lambda-calculus.

i. Syntax
Add a level of types above that used for general intersections. (We could syntactically restrict to intersections of compatible types, but then the syntax depends on the refinement relation)

ii. Base refinement relation

iii. Typing
For the new type level. We have exactly simply-typed lambda-calculus.

iv. Declarative Subsorting and sorting
Judgment: subsorting with the restriction to compatible sorts.
Judgment: sorting with the restriction to compatible sorts.
The rules for these judgements are the same as for general intersection types, perhaps with the refinement restrictions made explicit.
Theorem: finiteness of lattices up to equivalence, as per Freeman. Theorem: principal sorts, as per Freeman.

v. Algorithms
The previous algorithms and proofs do not need to be changed (although refinement restrictions could be made explicit).

3. More Concrete Views of Refinements
This chapter describes two more concrete views of refinement types. Later chapters will often make use of these views for intuitions and motivations. [This chapter could be merged into the previous one, depending on much I end up saying here.]

(a) Refinements as Properties

- A more commited view is that refinements include both elements of types, such as products, and elements of logical properties such as logical conjunction.
- Under this view refinements correspond to predicates over terms, entailment of properties is then subsorting, and intersection is logical conjunction.
- This is the view taken by Denney, who includes a full classical logic of properties as part of a refinement type system. Our goals somewhat different, because we want to consider a simple system for which checking can be done efficiently without requiring the programmer to provide proofs. We want the programmer to work with concepts similar to ordinary types,
- We can relate this view to the abstract view by translating a proof of a refinement typing judgment into a proof of a property. (If I can formally show this easily, I'll include it. Implement in Twelf?)
- We may not always be able translate a proof of a property into a refinement proof, since we want to allow simple refinement systems which are predictable and easily checkable, but not "complete" with respect to the provability of the corresponding properties.

(b) Refinements as Implicit Products and Coercion

- Another view of refinements is an extension of the implicit coercion interpretation of subtyping.
- Each base sort is translated to a base type, with coercions corresponding to subsorting, and intersections translated to products. Informally compare to work on Forsythe, which models intersection using pullbacks.
- Show this translation formally, and prove it correct with respect to reduction. (Implement in Twelf?)

4. Regular Refinements
A number of languages (ML, Haskell, ...) include definitions of named recursive sum types

(datatypes). Experience shows that this leads to a practical and managable type system which generally avoids the need to read or compare complicated type expressions. In this chapter we consider a way to define refinements of such types using regular tree grammars, following Freeman. We differ from Freeman in that we have a more explicit separation from the abstract sorting rules, and generally proceed at a more abstract level.

Following the view of subsorting as coercion, sorts defined by regular tree grammars correspond closely to allowing subtyping between the types defined by (inductive) datatype definitions [show an example, and continue it through the following discussion]. A common experience in programming large systems with datatypes is that sometimes there is a natural inclusion relationship between datatypes. Without subtyping, the programmer may write many functions which are essentially the same since they perform the same task on different (but related) datatypes. Writing explicit coercions between types helps a little here, but for recursive types such coercions are often extremely inefficient. Further, this does not work very well in some important cases such as where choosing a more accurate type for a function argument leads to a more accurate type for the result. In this case the programmer is forced to apply a "downcast" coercion to obtain a result in the more accurate type, which may fail at run time if due to an oversight by the programmer the actual result does not correspond to a value in the more accurate type. [Give example of the downcast from bits to nat, and an incorrect use to downcast "double epsilson" to nat, with the incorrect version of `double : bits -> bits` that just adds a zero to the end.]

A common alternative approach is to merge all related datatypes together, and consider the subtypes as "properties" or "invariants". However, in the absence of any mechanical way of checking these properties many runtime errors may occur due to the programmer incorrectly determining that a property holds, and thus failing to handle every possible case. Further, such properties are sometimes crucial to understanding a program, but are often poorly documented. Regular refinements can either be viewed as a way of checking such properties, or as a way of allowing a form of subtyping between datatypes. (These problems were the original inspiration for the more general notion of refinement types.)

[At the end of this chapter I could formally show how to translate a set of datasort declarations and a term to an equivalent set of datatype declarations and a term which uses explicit coercions. This extends the coercion interpretation in the previous chapter. I've sketched this, and it's actually slightly more interesting than one might think at first. I could probably also formally prove this translation correct. Similarly, I could show the translation to a set of properties, and how to translate a sorting derivation into a proof of a property.]

(a) Syntax for inductive datatypes
Programs:

```
local datatype a1 = c11 of A11 | ... | c1n of A1n
  and ...
  and datatype am = cm1 of Am1 | ... | cmn of Amn
   in M end
```

Terms: (extending simply typed lambda-calculus)

```
c M
case M of c1 x1 => M1 | ... | cn xn => Mn
```

This doesn't allow any form of recursion, even though we have inductive types, which is a little strange. Perhaps we want primitive recursion, although this issue is not so central to the point here.

(b) Typing
Extend the context with the definitions of the datatypes (or write this as a signature).
Judgment: Base type "a" hasclause "c of A".
A case must include all constructors in order.

(c) Reductions
Theorem: Subject reduction with datatypes.

(d) Datasort declaration syntax
We follow the syntax for datatype declarations, which is a restriction of Freeman. This is for simplicity, and also eliminates the problem of naming intermediate sorts in error messages. This is also close to what would be obtained if we tried to use separate datatypes for each datasort.

(e) Semantics as inductively defined subsets
Show the semantics, define subsorting as subset in the semantics, sort intersection as set intersection.
Theorem: This semantics satisfies the consistency conditions on subsorting and intersections in the simply-sorted lambda-calculus.

(f) Type theoretic interpretation (?)

Datasorts can be translated to recursive sums, with an empty type $\perp$ to allow a datasort to omit a value constructor which is in the datatype, and union types to allow a value constructor to be repeated. Then we can use the coinductive definition of subtyping for recursive types (as per Schwartzbach, ...) and the standard subtyping for union and intersection types. For inductive datasorts the subtyping that is obtained is equivalent to inclusion of regular tree grammars, for which good algorithms are known. [I'm reasonably sure this equivalence holds, but if not the reasons why would be very interesting.] This translation also yields a sorting algorithm for the language with datasorts (in the same sense that the type-theoretic interpretation of SML yields a type system for SML). Splitting naturally arises here, since under this interpretation we have that e.g.
$\texttt{nat} \leq \texttt{even} \vee \texttt{odd}$
corresponds to

$$\mu\alpha.\, 1 + \alpha \quad \leq \quad (\mu\beta.\, 1 + (\perp + \beta)) \vee (\mu\gamma.\, \perp + (1 + \gamma))$$

which is provable. We can similarly split `nat` modulo any $n \geq 1$, so there is no single best split to use during typechecking, and type checking is likely undecidable. This is one way to motivate the restriction to programmer defined refinements.

[We actually only need to have an empty refinement for each type, and unions of compatible types. For inductive datasorts, we actually only need a single empty refinement of the unit type 1, as this allows us to define empty refinements for each datasort. We still need compatible unions though.]

[We're not so interested in pure functions in this thesis, but this approach should allow us to treat effectful functions in datasorts via a type constructor for computations. I think the algorithm in my previous implementation properly implements the resulting subtyping.]

(g) Algorithms for subsorting
Mostly refer to work on algorithms for inclusion of regular tree grammars, but probably show an algorithm in our notation. Show how the algorithm for base types fits into the more abstract algorithm in the second chapter via the calculation of the lattice.

(h) Algorithms for lattice calculation
Start with an algorithm that compares all defined datasorts and intersections of any number of them. Improve on this by checking defined datasorts first, then adding intersections of pairs of the unique elements until all intersections are present. We can improve further, by avoiding comparisons of intersections of two sorts with supertypes of either sort, or with subtypes of both sorts. We can also avoid comparing r_3 to r_2 if we already found $r_1 < r_3$ and $r_2 < r_1$.

(i) Declarative sorting

The rule for value constructor application is straightforward. The rule for `case` requires some explanation. Perhaps the most declarative approach is to have an explicit split rule here (similar to the union left rule), and require splitting information to be included in the lattice. The splitting rule fits particularly well with the view of refinements as properties. An alternative is to carefully choose the strongest inversion principles, but I think in the presence of distributivity and the absence of effects splitting makes more sense. We need to give examples here to show cases where consideration of unions is natural once we decide to view datasort declarations as regular tree grammars, and where trying to avoid ever considering unions can lead to some unexpected results. E.g.

```
datatype a123 = c1 | c2 | c3
datasort a1 = c1 and a2 = c2

datatype b = d of a123
datasort b12 = d of a1 | d of a2

(*[ val f : a1 -> unit  &  a2 -> unit ]*)
fun f c1 = ()
  | f c2 = ()

(*[ val g : b12 -> unit ]*)
fun g (d x) = f x   (* should this succeed? *)

datasort a12 = c1 | c2
datasort b12' = d of a12
```

```
(*[ val g' : b12' -> unit ]*)
fun g' (d x) = f x   (* should this succeed? *)
```

We can argue that it is natural to consider inclusion of unions of regular tree grammars, and that we would expect these to act like union types.

(j) Calculating principal splits

We compare lattice elements with unions of sets of elements, and can make this more efficient in similar ways to the lattice creation.

(k) Sort checking
Show how to extend the sort checking algorithm, using the lattices and principal splits.

5. Refinements with Call-by-Value Effects
This chapter is essentially the ICFP paper using the framework in the previous section

6. Pattern Matching
Show a declarative and algorithmic formulation, and prove them equivalent. Show type preservation, perhaps via a translation. I don't think we want to consider pattern matching without effects, except for a few hints.

7. Extending to SML
Parameteric Polymorphism, Parameterized datasorts (mostly refer to Skalka), Modules.

8. Implementation

9. Experiments

10. Conclusion