Thesis Proposal:
Unifying Principles of Type Refinements

Joshua Dunfield
Carnegie Mellon University

December 2, 2003

Abstract

Traditional static type systems in the Hindley-Milner style are a useful means of guaranteeing type safety, in the form of type preservation and progress theorems, and of broadly specifying properties of functions (such as taking integers to lists of integers). In order to allow programmers to express stronger properties through static typing, I propose to develop and implement a rich type system that combines and extends work on datasort refinements, index refinements, intersection, and union types, and to show that it can check many interesting properties of functional programs that are difficult or impossible to check in conventional static type systems. Due to the richness of the type system, full type inference is undecidable, so a certain amount of annotation will be necessary. I intend to focus on properties of algebraic datatypes, such as list length or the set of keys mapped by a binary search tree. The type system's soundness is maintained in the presence of effects, but the properties are not *about* state. However, many of the techniques developed may be applicable to stateful properties as well.

1 Introduction

Conventional static type systems such as that of Standard ML [MTHM97] enable the expression of certain program invariants, allowing compilers to check those invariants at compile time. However, conventional type systems are too coarse-grained to check many desirable properties, leading to the development of *refined* type systems such as the *datasort refinements* of Freeman, Davies, and Pfenning [FP91, Fre94, DP00], and the *index refinements* of Xi and Pfenning [XP99, Xi98]. Both systems refine the *simple types* of Hindley-Milner type systems.

My work unifies and extends the datasort and index refinement systems, which have distinct mechanisms for combining properties: intersection types for the datasort refinements, universal and existential quantification and subset sorts for the index refinements. I provide unified mechanisms: *definite types* (including intersections and universal quantification) for the conjunction of properties, *indefinite types* (including existential quantification) for the disjunction of properties. As Xi found [Xi98], determining the scope of the elimination rules for indefinite types is not straightforward; I handle this through careful formulations of both my type system and my variant of let-normal or A-normal form.

Because pure type inference is, variously, undesirable or impossible in these refinement systems, the programmer must write type annotations at certain points in the program.

In this proposal, I start by discussing background to my work, particularly the key concepts of datasort refinements, index refinements, and intersection and union types. Next, I outline the broad contours of my approach and discuss several technical issues in more depth. After examining some related work, I explain what work I have already done and what I plan to do.

2 Background

As with conventional static type systems (Standard ML, Haskell, Java), I view types as describing program properties that can be checked at compile time (in contrast to *dynamic typing* where checks are performed at runtime, a shortcoming only partially addressed by the *soft typing* systems discussed below). My type system "pushes the envelope" of expressible properties by incorporating several features not found in conventional type systems:

- datasort refinements
- intersection types
- index refinements
- union types

2.1 Datasort refinements and intersection types

Datasort refinements serve to check properties that can be defined by *regular tree grammars*, a generalization of regular grammars from strings to trees [CDG+97]. Regular tree grammars are recognized by regular tree automata; automata recognizing ordinary regular languages on strings can be seen as an instance of regular tree automata, where the trees in the regular tree grammar all have the shape of a line. A very simple example is the property of a list (of integers) being of even length or odd length:

- The empty list Nil is of even length;
- If t is of even length then $\mathsf{Cons}(h, t)$ is of odd length;
- If t is of odd length then $\mathsf{Cons}(h, t)$ is of even length;
- If t is of unknown length then $\mathsf{Cons}(h, t)$ is of unknown length.

Notice that we can express the property of Cons using only the property of its argument t. Choosing to write even and odd to denote the properties of having even or odd length and list to denote the property of having unknown length, we can write the types of the constructors Nil and Cons as

$$\begin{aligned} \mathsf{Nil} &: \mathsf{even} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{even} \to \mathsf{odd} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{odd} \to \mathsf{even} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{list} \to \mathsf{list} \end{aligned}$$

But this doesn't make much sense: we have three different types for Cons. A natural type-theoretic expression of this is *intersection types*: $A \wedge B$ denotes the intersection of the types A and B, that is, the conjunction of the properties expressed by A and B.

$$\begin{aligned} \mathsf{Nil} &: \mathsf{even} \\ \mathsf{Cons} &: (\mathsf{int} * \mathsf{even} \to \mathsf{odd}) \\ &\quad \wedge (\mathsf{int} * \mathsf{odd} \to \mathsf{even}) \\ &\quad \wedge (\mathsf{int} * \mathsf{list} \to \mathsf{list}) \end{aligned}$$

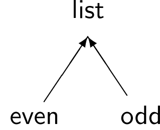
A simple function of intersection type is *tail*:

$$\begin{aligned} \mathit{tail} \;=\; \lambda x.\, &\mathbf{case}\ x\ \mathbf{of} \\ &\quad \mathsf{Nil} \Rightarrow \mathbf{raise}\ \mathit{Error} \\ &\quad \mid \mathsf{Cons}(h, t) \Rightarrow t \end{aligned}$$

It is easy to see that *tail* has type

$$tail \;:\; (\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{list} \to \mathsf{list})$$

Implicit in this discussion is that every value of type even (resp. odd) is known to be a value of type list. Formally, we assume the existence of a *subsort relation* $\preceq$ defining a partial order on datasorts. In this case our subsort relation is the reflexive closure of $\{(\mathsf{even}, \mathsf{list}), (\mathsf{odd}, \mathsf{list})\}$, represented by the diagram



In practice, the typechecker determines the constructor types and subsort relation from a sequence of *sort declarations*:

$$\textbf{datasort}\ \mathsf{even} = \mathsf{Nil} \mid \mathsf{Cons}\ \textbf{of}\ \mathsf{int} * \mathsf{odd}$$
$$\textbf{and}\ \mathsf{odd} = \mathsf{Cons}\ \textbf{of}\ \mathsf{int} * \mathsf{even}$$

Intersection types and datasort refinements Intersection types [CDCV81] were first incorporated into a practical language by Reynolds [Rey96], who used them to encode features such as operator overloading, and proved that typechecking intersection types is PSPACE-hard. The notion of datasort refinement combined with intersection types was introduced by Freeman and Pfenning [FP91], who showed that full type inference was decidable under the *refinement restriction* ($A \wedge B$ permitted only if A and B are refinements of the same type) by using techniques from abstract interpretation. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning [DP00], who introduced a value restriction on intersection introduction, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm. Davies implemented datasort refinements [Dav97, Dav04] as an extension to the ML Kit [ML], a Standard ML compiler.

2.2 Index refinements and union types

Index refinements refine types by indices drawn from a decidable constraint domain. The theory [Xi98] is parametric in the constraint domain, but (by far) the most extensively used domain is the integers with linear inequalities, which permits one to express quite useful properties such as the size of data structures. Continuing with our list example, the types of Nil and Cons in an index refinement setting are

$$\mathsf{Nil} \;:\; \mathsf{list}(0)$$
$$\mathsf{Cons} \;:\; \Pi a{:}\mathcal{N}.\ \mathsf{int} * \mathsf{list}(a) \to \mathsf{list}(a+1)$$

Here $\Pi a{:}\mathcal{N}.$ universally quantifies over an index a having *index sort* $\mathcal{N}$ (the natural numbers), so the type of Cons should be read "for any natural number a, Cons takes an integer and a list of length a and returns a list of length $a+1$". The index-refined type of the function *tail* (defined previously) is

$$tail \;:\; \Pi a{:}\{a{:}\mathcal{N} \mid a > 0\}.\ \mathsf{int} * \mathsf{list}(a) \to \mathsf{list}(a-1)$$

In this type we use the *subset sort* $\{a{:}\mathcal{N} \mid a > 0\}$ read "a is a natural number such that a is greater than 0". Since the domain of the type is $\mathsf{int} * \mathsf{list}(a)$ where $a > 0$, *tail* cannot be applied to an empty list.

To express the type of a function such as *filter*, where *filter* $f\ l$ returns the elements of l for which f returns true, we need existential quantification $\Sigma b{:}\gamma.\ B$. Index refinements are not powerful enough to encode the precise length returned.

$$filter : \Pi a{:}\mathcal{N}.\ (\mathsf{int} \to \mathsf{bool}) \to \mathsf{list}(a) \to \Sigma b{:}\mathcal{N}.\ \mathsf{list}(b)$$

The binary union $A \vee B$ expresses the disjunction of the properties specified by A and B. A type $\Sigma b{:}\gamma.\, B$ can be viewed as infinite (for infinite index domains) *union types*; for example, $\Sigma b{:}\mathcal{N}.\, \mathsf{list}(b)$ corresponds informally to

$$\mathsf{list}(0) \vee \mathsf{list}(1) \vee \mathsf{list}(2) \vee \cdots$$

We call existential quantification and union types *indefinite types*. In contrast to existential quantification, union types may be more of a convenience than a necessity: there seems to always be an alternative way to write the type, for example, $(A \vee B) \to C$ can be written $(A \to C) \wedge (B \to C)$ (but this is only a conjecture based on limited experience). The inclusion of union types is justified by symmetry with intersections and by the similarity of the technical realizations of union types and Σ types.

Dependent type systems The theoretical root of index refinements is the notion of *dependent type* found in type systems such as Nuprl [CAB+86]. There, dependent products $\Pi x{:}A.\, B$ and dependent sums $\Sigma x{:}A.\, B$ roughly correspond to the universal and existential quantifiers over indices; however, instead of drawing x from a restricted index domain, dependent products and sums draw x from terms of type A. This is extremely powerful but also undecidable, making it appropriate for theorem proving systems but not for static typechecking.

DML: index refinements Xi formulated a bidirectional type system [Xi98] with index refinements for a variant of ML and implemented it as an extension to Caml Light [Cam]. He showed a number of applications (based on the integer constraint domain he implemented), including array bounds check elimination. The technical formulation of index refinements in DML differs somewhat from mine; these differences are discussed briefly in Section 4.

Termination checking Given a recursive function, if we can formulate a well-founded metric of its arguments such that the metric strictly decreases across recursive calls, those recursive calls do not cause nontermination. Xi [Xi01, Xi02] showed that a relatively simple extension of the machinery of index refinements suffices to statically check termination.

Union types Pierce [Pie91b] gave examples of programming with intersection and union types in a pure λ-calculus using a typechecking mechanism that relied on syntactic markers. The first systematic study of unions in a type assignment framework by Barbanera et al. [BDCd95] identified a number of problems, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted.

3 Thesis statement

With the above background, I can state my thesis:

> **A rich type system with datasort and index refinement properties, where such properties are combined through intersection and union types, is a practical means of statically checking interesting properties of functional programs that are difficult or impossible to check in conventional static type systems.**

4 Technical overview

Here I discuss key technical aspects of my approach and contrast it to previous work on the refinement systems in isolation. I necessarily omit a full presentation of the material; most of the omitted details can be found in [Dun02, DP03, DP04]. The setting is a small functional language

$$
\begin{array}{rrll}
\text{Types} & A, B, C ::= & \mathbf{1} \mid A \to B \mid A * B & \\
& & \mid \delta(i) & \text{—Refinement by datasort } \delta \text{ and index } i \\
& & \mid A \wedge B \mid \Pi a{:}\gamma.\, A & \\
& & \mid A \vee B \mid \Sigma a{:}\gamma.\, A & \\
\text{Terms} & e ::= & x \mid u \mid \lambda x.\, e \mid e_1\, e_2 \mid \textbf{fix}\, u.\, e & \\
& & \mid () \mid (e_1, e_2) \mid \textbf{fst}(e) \mid \textbf{snd}(e) & \\
& & \mid c(e) \mid \textbf{case}\; e\; \textbf{of}\; ms & \\
\text{Matches} & ms ::= & \cdot \mid c(x) \Rightarrow e \mid ms & \\
\text{Values} & v ::= & x \mid \lambda x.\, e \mid () \mid (v_1, v_2) & \\
\text{Eval. contexts} & E ::= & [\,] \mid E(e) \mid v(E) & \\
& & \mid (E, e) \mid (v, E) \mid \textbf{fst}(E) \mid \textbf{snd}(E) & \\
& & \mid c(E) \mid \textbf{case}\; E\; \textbf{of}\; ms &
\end{array}
$$

$$
\frac{e' \mapsto_R e''}{E[e'] \mapsto E[e'']}
\qquad
\begin{array}{rcl}
(\lambda x.\, e)\, v & \mapsto_R & [v/x]\, e \\
\textbf{fix}\, u.\, e & \mapsto_R & [\textbf{fix}\, u.\, e \,/\, u]\, e
\end{array}
\qquad
\begin{array}{rcl}
\textbf{fst}(v_1, v_2) & \mapsto_R & v_1 \\
\textbf{snd}(v_1, v_2) & \mapsto_R & v_2
\end{array}
$$

$$
\textbf{case}\; c(v)\; \textbf{of}\; \ldots c(x) \Rightarrow e \ldots \mapsto_R [v/x]\, e
$$

Figure 1: Syntax and semantics of the core language

with a call-by-value semantics (Figure 1). The semantics does not include mutable references, but the type system has been designed with mutable references in mind. After discussing certain major design decisions, I explain bidirectionality and how intersection and union types are handled, discuss a novel form of typing annotation, and cite key theorems proved in the papers [Dun02, DP03, DP04].

4.1 Rationale

The basic goal of this work is to build a single type system capable of expressing both the invariants expressible through datasort refinements *and* those expressible through index refinements. In this section, I explain and justify several important aspects of the design of the system:

- the inclusion of two partly overlapping refinement mechanisms: datasorts and indices
- the use of typing annotations and a bidirectional type system, instead of full type inference

4.1.1 Overlapping refinement mechanisms

A natural question to ask is whether one variety of refinement is strictly more expressive than the other. It is quite clear that there are properties expressible by index refinements, such as the precise length of a list, that cannot be expressed by datasort refinements: The length of a list is a natural number, a member of an infinite set; a finite lattice of datasorts can only distinguish among a finite number of subsets of that infinite set.

However, the question of whether index refinements subsume datasort refinements is harder to answer. Often, at least, one can easily encode the datasort as an index from a small domain, a simple example being bool indexed by 0 or 1, which is as expressive as the false/true datasort refinement. So why not use index refinements exclusively, avoiding the complications of intersection types?

- In many situations, datasort refinements seem more natural to write. But this is not so compelling: perhaps the system could regard datasort refinements as syntactic sugar for index refinements by small integers.

- A straightforward encoding of datasort refinements into index refinements leads to generated constraints that are too complex (with rampant existential quantification) for current implementations of index refinements (Xi's deCaml).
- Davies' implementation [Dav97, Dav04] has clearly demonstrated the practicality of a datasort refinement system *not* based on index refinements.

It is difficult to rule out the possibility of successfully encoding datasorts as indices: we would have to prove that no appropriate constraint domain exists. Thus, while I cannot show it is impossible to use index refinements alone, I see no compelling reason not to build on the successful datasort refinement work of Davies and Pfenning. Therefore, I commit myself to a system with both datasort and index refinements as primitive mechanisms.

4.1.2 Type inference

Type inference is decidable for datasort refinements; in fact, the original work of Freeman and Pfenning [FP91] on datasorts was based on inference (implemented through abstract interpretation) and was reasonably efficient. However, type inference is undecidable for index refinements, so if we want both varieties of refinement we must abandon full type inference and accept that we will have to write type annotations. But to make a virtue out of the necessity of type annotations, consider:

- Type annotations provide useful documentation—more so in a refined type system, as the properties are more interesting!
- Type annotations circumscribe the meaning of components (such as functions and modules), so that these components cannot be used in ways that are *well-defined* but *unintended*. For example, one never wants to apply *tail* to an empty list, but its behavior is well-defined (it raises an exception and 'raise' expressions are always well typed) and a perfectly good type can be inferred. This issue becomes especially important in large programs, and was a key factor in Davies and Pfenning's decision to *not* use inference techniques in their work on datasort refinements.[1]
- Type inference is already undecidable for realistic languages; for example, at the module level of Standard ML, types must be written out.
- The past work on the individual refinement systems suggests that the amount of annotation needed is modest [XP99, Dav04], and certainly not as clumsy as conventional explicitly typed languages such as Java and C++.

4.2 Bidirectional typechecking

My type system is *bidirectional*: instead of a single form of typing judgment $\Gamma \vdash e : A$, the system has two judgment forms:

Synthesis:	$\Gamma \vdash e \uparrow A$	In context Γ, e synthesizes type A
Checking:	$\Gamma \vdash e \downarrow A$	In context Γ, e checks against type A

The first form, synthesis, is conceptually similar to the traditional $\Gamma \vdash e : A$ form: Given a context Γ and term e, generate (synthesize) some type A. The second form, checking, differs in that all information—Γ, e, and A—is assumed, and the task is to confirm that e checks against A.

[1] Inference may be desirable in cases such as nested functions, which cannot really be considered as "components". However, my approach at present makes no attempt to distinguish such cases.

Here is an example to give part of the flavor of bidirectional typechecking (and typechecking with intersection types): the *tail* function:

$$\begin{aligned} \textit{tail} \;=\; & \lambda x.\, \textbf{case}\ x\ \textbf{of} \\ & \quad \mathsf{Nil} \Rightarrow \textbf{raise}\ \textit{Error} \\ & \quad \mid \mathsf{Cons}(h, t) \Rightarrow t \end{aligned}$$

Our job is to check this function against $(\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{list} \to \mathsf{list})$. To check a value against an intersection type, we check it against each conjunct, so we start by checking against $\mathsf{even} \to \mathsf{odd}$.

$$\vdash \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \ldots \downarrow \mathsf{even} \to \mathsf{odd}$$

Like a conventional type system, we assume the λ-bound variable x is of type even (the domain of the function space), and check the body against type odd (the codomain).

$$x{:}\mathsf{even} \vdash \textbf{case}\ x\ \textbf{of}\ \ldots \downarrow \mathsf{odd}$$

To check a **case** expression we *synthesize* a type for the subject expression, namely x, then check each possible arm under appropriate assumptions. We have $x{:}\mathsf{even}$ in our context, so we can synthesize type even for x.

$$\frac{\dfrac{}{x{:}\mathsf{even} \vdash x \uparrow \mathsf{even}} \qquad \begin{array}{l} x{:}\mathsf{even} \vdash \mathsf{Nil} \Rightarrow \textbf{raise}\ \textit{Error} \downarrow_{\mathsf{even}} \mathsf{odd} \\ x{:}\mathsf{even} \vdash \mathsf{Cons}(h,t) \Rightarrow t \quad \downarrow_{\mathsf{even}} \mathsf{odd} \end{array}}{x{:}\mathsf{even} \vdash \textbf{case}\ x\ \textbf{of}\ \ldots \downarrow \mathsf{odd}}$$

The subscripted judgment $\ldots \vdash \ldots \downarrow_{\mathsf{even}} \mathsf{odd}$ means that the branch is checked with the assumption that the expression cased upon has type even. Clearly **raise** *Error* checks against odd, so consider the second branch. Here we must show $t \downarrow \mathsf{odd}$. Recall the type of Cons:

$$\begin{aligned} \mathsf{Cons}\ :\ & (\mathsf{int} * \mathsf{even} \to \mathsf{odd}) \\ & \wedge (\mathsf{int} * \mathsf{odd} \to \mathsf{even}) \\ & \wedge (\mathsf{int} * \mathsf{list} \to \mathsf{list}) \end{aligned}$$

We know we have an even list, so the appropriate conjunct is $(\mathsf{int} * \mathsf{odd} \to \mathsf{even})$.[2] So we assume h has type int and t has type odd, and check the body of the case arm (which is just t) against odd:

$$x{:}\mathsf{even}, h{:}\mathsf{int}, t{:}\mathsf{odd} \vdash t \downarrow \mathsf{odd}$$

In fact, we check t against odd in two steps: first we synthesize odd, then apply a subsumption typing rule. $\Gamma \vdash \mathsf{odd} \leq \mathsf{odd}$ is a subtyping judgment, discussed below.

$$\frac{\dfrac{}{t{:}\mathsf{odd} \vdash t \uparrow \mathsf{odd}} \qquad \Gamma \vdash \mathsf{odd} \leq \mathsf{odd}}{x{:}\mathsf{even}, h{:}\mathsf{int}, t{:}\mathsf{odd} \vdash t \downarrow \mathsf{odd}}$$

Ultimately, the derivation looks like

$$\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{even} \vdash x \uparrow \mathsf{even}} \qquad \dfrac{\vdots}{x{:}\mathsf{even} \vdash \mathsf{Nil} \Rightarrow \textbf{raise}\ \textit{Error} \downarrow_{\mathsf{even}} \mathsf{odd}} \qquad \dfrac{\dfrac{\dfrac{}{t{:}\mathsf{odd} \vdash t \uparrow \mathsf{odd}} \quad \Gamma \vdash \mathsf{odd} \leq \mathsf{odd}}{x{:}\mathsf{even}, h{:}\mathsf{int}, t{:}\mathsf{odd} \vdash t \downarrow \mathsf{odd}}}{x{:}\mathsf{even} \vdash \mathsf{Cons}(h,t) \Rightarrow t \downarrow_{\mathsf{even}} \mathsf{odd}}}{x{:}\mathsf{even} \vdash \textbf{case}\ x\ \textbf{of}\ \ldots \downarrow \mathsf{odd}}}{\vdash \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \ldots \downarrow \mathsf{even} \to \mathsf{odd}}$$

[2] It might seem that we also need to look at $(\mathsf{int} * \mathsf{list} \to \mathsf{list})$, since $\mathsf{even} \preceq \mathsf{list}$. However, if $\mathsf{int} * \mathsf{list} \to \mathsf{list}$ had been applied we could not know that the list is even. But we do know it is even, so $(\mathsf{int} * \mathsf{list} \to \mathsf{list})$ is irrelevant.

4.2.1 Other formulations of bidirectionality

I currently use a form of bidirectionality based on the principle that introduction rules check and elimination rules synthesize.[3] For example, the rules involving pairs $A * B$ are

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

This is the smallest reasonable set of rules. It would be quite pointless to replace $(*\mathrm{E}_1)$ and $(*\mathrm{E}_2)$ with checking rules: if we check $\mathbf{fst}(e)$ against A we are not given B, so we must synthesize e's type $A * B$, and if we can synthesize $A * B$ we have A and should avoid the possible need for an annotation by synthesizing the type of $\mathbf{fst}(e)$. Moreover, if we tried replacing $(*\mathrm{I})$ with a rule synthesizing $A_1 * A_2$, we would often have to write annotations on the components e_1 and e_2.

So, while there is no point in having checking forms of the elimination rules, it could be useful to have a synthesis form of the introduction rule:

$$\frac{\Gamma \vdash e_1 \uparrow A_1 \quad \Gamma \vdash e_2 \uparrow A_2}{\Gamma \vdash (e_1, e_2) \uparrow A_1 * A_2}\ (*\mathrm{I}\text{-}\uparrow)$$

Such a rule would allow the system to typecheck terms such as

$$\mathbf{case}\ (x, y)\ \mathbf{of}\ \ldots$$

Here x and y are variables and therefore synthesize. If we have rule $(*\mathrm{I}\text{-}\uparrow)$, the pair (x, y) also synthesizes. Without this rule, we would have to put a typing annotation on (x, y).

Xi's index refinement system [Xi98] includes both $(*\mathrm{I})$ and $(*\mathrm{I}\text{-}\uparrow)$. In fact, his system includes every bidirectional variation that would not be obviously useless (in the way that a checking form of $(*\mathrm{E}_1)$ is useless), leading to a type system that is slightly more convenient to use but slightly more complicated. I believe that adapting my system to Xi's style of bidirectionality would be straightforward, but to avoid premature optimization I plan to do so only if the lack of rules such as $(*\mathrm{E}_1\text{-}\uparrow)$ is burdensome in the course of my experiments.

Pierce and Turner's *local type inference* [PT98] is another bidirectional system with synthesis and checking judgments, somewhat similar to the present system (and to Xi's). Their language has subtyping and impredicative polymorphism, making full type inference undecidable. In order to handle parametric polymorphism without using nonlocal methods such as unification, they infer type arguments to polymorphic functions, which seems to substantially complicate matters. Hosoya and Pierce [HP99] further discuss this style, particularly its effectiveness in achieving a reasonable number of annotations.

4.3 Intersection types

A value v has type $A \wedge B$ if it has type A and type B. Because this is an introduction form, we proceed by *checking* v *against* A and B. Conversely, if e has type $A \wedge B$ then it must have both type A and type B, proceeding in the direction of synthesis. The introduction rule $(\wedge\mathrm{I})$ is restricted to values because its general form for arbitrary expressions e is unsound in the presence of mutable references in call-by-value languages [DP00].

$$\frac{\Gamma \vdash v \downarrow A \quad \Gamma \vdash v \downarrow B}{\Gamma \vdash v \downarrow A \wedge B}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow A}\ (\wedge\mathrm{E}_1) \qquad \frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow B}\ (\wedge\mathrm{E}_2)$$

The system's subtyping rules are designed following the well-known principle that $A \leq B$ only if any (closed) value of type A also has type B. Thus, whenever we must check if an expression e has type B we are safe if we can synthesize a type A and $A \leq B$:

[3] This principle is discussed further in [DP04].

$$P ::= \bot \mid i \doteq j \mid \ldots \qquad \Gamma ::= \cdot \mid \Gamma, x{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P$$

$$\overline{\cdot} = \cdot \qquad \overline{\Gamma, x{:}A} = \overline{\Gamma} \qquad \overline{\Gamma, a{:}\gamma} = \overline{\Gamma}, a{:}\gamma \qquad \overline{\Gamma, P} = \overline{\Gamma}, P$$

Figure 2: Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

$$\frac{\Gamma \vdash e \uparrow A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e \downarrow B}\ \textbf{(sub)}$$

The rules for intersection are:

$$\frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ (\wedge\textbf{R})$$

$$\frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\textbf{L}_1) \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\textbf{L}_2)$$

4.4 Refined Datatypes

As in the informal examples above, we write $\delta(i)$ for the type of values having datasort δ and index i. To accommodate index refinements, the context Γ allows index variables a, b and propositions P as well as program variables. Because the program variables are irrelevant to the index domain, we can define a *restriction function* $\overline{\Gamma}$ that yields its argument Γ without program variable typings (Figure 2). No variable may be declared twice in Γ, but ordering is now significant because of dependencies.

Our formulation, like Xi's, requires only a few properties of the constraint domain: There must be a way to decide a consequence relation $\overline{\Gamma} \models P$ whose interpretation is that given the index variable typings and propositions in $\overline{\Gamma}$, the proposition P must hold. Because we have both universal and existential quantifiers over elements of the constraint domain, the constraints must remain decidable in the presence of quantifiers, though we have not encountered quantifier alternations in our examples. There must also be a relation $i \doteq j$ denoting index equality, and a judgment $\overline{\Gamma} \vdash i : \gamma$ whose interpretation is that i has *index sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have index sorts; terms and indices are distinct. The proof of safety in [DP03] requires that $\models$ be a consequence relation, that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that $\models$ and $\vdash$ have expected substitution and weakening properties [Dun02].

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. For the moment, think of an index sort $\mathcal{N}$ of natural numbers, with arithmetic operations $+, -, *$. Then $\overline{\Gamma} \models P$ is decidable provided the equalities in P are linear.

The subtyping rule for datatypes checks the datasorts δ_1, δ_2 and (separately) the indices i, j; to maintain reflexivity and transitivity of subtyping, we require $\preceq$ to be reflexive and transitive.[4]

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

We assume the constructors c are typed by a judgment $\overline{\Gamma} \vdash c : A \rightarrow \delta(i)$ where A is any type and $\delta(i)$ is some refined type. The type $A \rightarrow \delta(i)$ need not be unique; indeed, a constructor should often have more than one refined type. The rule for constructor application is

[4] The system cannot deduce the emptiness of refinements such as $\mathsf{pos}(0)$ where pos denotes a bitstring of strictly positive 2's-complement value and the index refinement 0 denotes a bitstring of length 0. Experience will show if this inability makes a difference in practice.

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

To derive $\Gamma \vdash$ **case** e **of** $ms \downarrow B$, we check that all the matches in ms check against B, under a context appropriate to each arm; this is how propositions P arise. The context Γ may be contradictory ($\overline{\Gamma} \models \bot$) if the case arm can be shown to be unreachable by virtue of the index refinements of the constructor type and the case subject. In order to not typecheck unreachable arms, we have

$$\frac{\overline{\Gamma} \models \bot}{\Gamma \vdash e \downarrow A} \ (\mathbf{contra})$$

We also do not check case arms that are unreachable by virtue of the *datasort* refinements. For a complete accounting of how we type **case** expressions and constructors, see [Dun02].

The typing rules for universal index quantification Π are

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

The index variable a added to the context must be new, which can always be achieved via renaming. As for intersections, the introduction rule is restricted to values in order to maintain type preservation in the presence of effects.

One potentially subtle issue with the introduction rule is that v cannot reference a in an internal type annotation, because that would violate α-conversion: one could not safely rename a to b in $\Pi a{:}\gamma.\, A$, which is the natural scope of a. I discuss this further in the section on *contextual typing annotations*.

The subtyping rules for Π are

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

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

As written, in (ΠL) and (ΠE) we must guess the index i; in practice, we would plug in a new existentially quantified index variable and continue, using constraint solving to determine i. Thus, even if we had no existential types Σ in the system, the solver for the constraint domain would have to allow existentially quantified variables.

4.5 Union types

Union types $A \vee B$ and existential types $\Sigma a{:}\gamma.\, A$ quantifying existentially over an index a of sort γ are substantially harder to typecheck than intersection types and universal index quantification. In particular, the elimination rules are difficult to formulate in a sound and satisfactory way. The thinking leading to the rules below is explained in [DP03]; in particular, several apparently plausible variations are shown to be unsound.

On values, the binary indefinite type $A \vee B$ is simply a union in the ordinary sense: if $v : A \vee B$ then either $v : A$ or $v : B$. The introduction rules directly express the simple logical interpretation, again using checking for the introduction form.

$$\frac{\Gamma \vdash e \downarrow A}{\Gamma \vdash e \downarrow A \vee B} \ (\vee\mathrm{I}_1) \qquad \frac{\Gamma \vdash e \downarrow B}{\Gamma \vdash e \downarrow A \vee B} \ (\vee\mathrm{I}_2)$$

No restriction to values is needed for the introductions, but, dually to intersections, the elimination must be restricted. A sound formulation of the elimination rule in a type assignment form [DP03] without a syntactic marker requires an evaluation context E around the subterm e' of union type.

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

In this rule, we no longer move from terms to their immediate subterms, but when type-checking e we may have to decompose it into an evaluation context E and subterm e'. Using the analysis and synthesis judgments we have

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

Here, if we can synthesize a union type for e'—which is in evaluation position in $E[e']$—and check $E[x]$ and $E[y]$ against C, assuming that x and y have type A and type B respectively, we can conclude that $E[e']$ checks against C.

The subtyping rules are standard and dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \quad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B}\ (\vee\mathbf{L})$$

$$\frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2}\ (\vee\mathbf{R}_1) \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2}\ (\vee\mathbf{R}_2)$$

For existential quantification, the introduction rule presents no difficulties, and proceeds using the analysis judgment.

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

For the elimination rule, we follow $(\vee\mathrm{E})$:

$$\frac{\Gamma \vdash e' \uparrow \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash E[x] \downarrow C}{\Gamma \vdash E[e'] \downarrow C}\ (\Sigma\mathrm{E})$$

Again, there is a potentially subtle issue: the index variable a must be new and cannot be mentioned in an annotation in E.

The subtyping for Σ is dual to that of Π.

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

We now have rules $(\vee\mathrm{E})$ and $(\Sigma\mathrm{E})$ for binary and infinitary indefinite types. It turns out that there is a sensible unary version of these rules:

$$\frac{\Gamma \vdash e' \uparrow A \qquad \Gamma, x{:}A \vdash E[x] \downarrow C}{\Gamma \vdash E[e'] \downarrow C}\ (\textbf{direct})$$

One might expect this rule to be admissible. However, due to the restriction to evaluation contexts, it is not. As a simple example, consider

$$\begin{array}{rl} \textit{append} & : \ \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(a+b) \\ \textit{filterpos} & : \ \Pi n{:}\mathcal{N}.\, \mathsf{list}(n) \to \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ & \vdash \textit{filterpos}\ [\ldots] \uparrow \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ \text{Goal:} & \not\vdash \textit{append}\ [42]\ (\textit{filterpos}\ [\ldots]) \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k) \end{array}$$

where [42] is shorthand for $\mathsf{Cons}(42, \mathsf{Nil})$ and [...] is some literal list. Here we cannot derive the goal, because we cannot introduce the k on the type checked against. To do so, we would need to introduce the index variable m representing the length of the list returned by *filterpos* [...], and use $m+1$ for k. But *filterpos* [...] is not in evaluation position, because *append* [42] will need

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

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

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

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

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

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

Figure 3: Subtyping

to be evaluated first. However, *append* [42] synthesizes only type $\Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(1 + b)$, so we are stuck. But using rule (direct) we can collapse *append* [42] into a fresh variable x. Since x is a value, the subterm *filterpos* [...] is now in evaluation position, allowing us to use (ΣE).

While ($\vee$E), (ΣE), and (direct) are satisfactory on paper, a straight implementation would be excessively nondeterministic: a given term e might have many possible evaluation contexts E such that $e = E[e']$, and the choice of which contexts to use and in what order is left open. I aim to resolve this by formulating a new variant of let-normal form, which is akin to continuation-passing style (CPS): Intermediate results are named, and the evaluation order is made explicit by inserting **let** expressions. However, unlike CPS, let-normal form does not introduce continuations. (Hence, let-normal form is sometimes called "two-thirds CPS".) Restricting the above rules to terms of the form **let** $x = e'$ **in** $E[x]$ forces the typechecker to proceed through the term in evaluation order. Care must be taken to ensure that the system so restricted is complete, in the sense of preserving the well-typing of programs.[5] Note that Xi also used a translation to let-normal form, but his particular formulation interacts with bidirectionality in unfortunate ways—certain programs that *are* well typed as written are *not* well typed after translation.

4.6 Properties of Subtyping

The subtyping rules, shown in Figure 3, work in a natural way for intersections, unions, and index quantification (as well as a greatest type $\top$ and a least type $\bot$). At the level of the refinements, the judgment passes to the subsorting and index satisfaction relations:

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

Decidability, reflexivity, and transitivity of subtyping are straightforward to prove [Dun02, DP03].

4.7 Contextual typing annotations

Some terms require annotations in my system. For example, $(\lambda x.\, x)()$ neither synthesizes nor checks against a type. This is because the function part of an application must synthesize a

[5] This work is almost complete: the transformation has been precisely formulated and the appropriate proofs are nearly finished.

type, but there is necessarily no rule for $\lambda x.\, e$ to synthesize a type. However, the standard form of annotation $(e : A)$ does not suffice in this setting.

First, when checking a function $\lambda x.\, e$ against an intersection such as $(A_1 \to B_1) \wedge (A_2 \to B_2)$, annotations inside the function body e may need to vary depending on which branch is being checked. Pierce [Pie91a] and Reynolds [Rey96] addressed this problem by allowing a function to be annotated with a list of alternative types; the typechecker chooses the right one. Davies followed this approach in his datasort refinement checker, allowing a term to be annotated with $(e : A, B, \dots)$. This notation is easy to use and effective but introduces additional nondeterminism, since the typechecker must guess which type to use.

Second, the annotations may need to mention index variables, but in the absence of an explicit marker for Π-introduction, the scope of such variables is unclear. For example, when checking $\lambda x.\, e$ against $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a+1)$, should a be meaningful when used in an annotation inside e? The answer should be no: the proper scope of a is limited to the type $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a+1)$, and we should be able to consistently rename a within this type without making the term ill-typed. Xi addressed this problem through a term-level abstraction over index variables, $\Lambda a.e$, to mirror universal index quantification $\Pi a{:}\gamma.\, A$ [Xi98]. But this violates the basic principle of property types that the term should remain unchanged, and fails in our setting due to the presence of intersections. For example, we would expect the reverse function on lists, *rev*, to satisfy

$$\begin{aligned} \mathit{rev} \ &: \ \big(\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a)\big) \\ &\wedge \big((\Sigma b{:}\mathcal{N}.\, \mathsf{list}(b)) \to \Sigma c{:}\mathcal{N}.\, \mathsf{list}(c)\big) \end{aligned}$$

but the first component of the intersection demands a term-level index abstraction, while the second does not tolerate one.

We address these problems by a method that builds upon the notation of comma-separated alternatives. The essential idea is to allow a context to appear in the annotation along with each type:

$$e \ ::= \ \dots \mid (e : \Gamma_1 \vdash A_1, \dots, \Gamma_n \vdash A_n)$$

where each context Γ_k declares the types of some, but not necessarily all, free variables in e.

We can think of such an annotated term as follows: if $\Gamma_k \vdash e \downarrow A_k$ then $\Gamma \vdash (e : \Gamma_1 \vdash A_1, \dots, \Gamma_n \vdash A_n) \uparrow A_k$ if the current assumptions in Γ validate the assumptions in Γ_k. For example, the second judgment below is not derivable, since $x{:}\mathsf{odd}$ does not validate $x{:}\mathsf{even}$ (because $\mathsf{odd} \not\leq \mathsf{even}$).

$$\begin{aligned} x{:}\mathsf{even} &\vdash \big((\lambda y.\, \mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ &\qquad\qquad\qquad\qquad\quad x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}\big) \uparrow \mathbf{1} \to \mathsf{odd} \\ x{:}\mathsf{odd} &\not\vdash \big((\lambda y.\, \mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ &\qquad\qquad\qquad\qquad\quad x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}\big) \uparrow \mathbf{1} \to \mathsf{odd} \end{aligned}$$

In practice, this should significantly reduce the nondeterminism associated with type annotations in the presence of intersection. Moreover, we can write a suitable annotation As in

$$\vdash \lambda x.\, \big((\lambda z.\, x) : As\big)\,()\downarrow \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a)$$

through a locally declared index variable b:

$$\lambda x.\, \big((\lambda z.\, x) : (b{:}\mathcal{N}, x{:}\mathsf{list}(b) \vdash \mathbf{1} \to \mathsf{list}(b))\big)$$

Now, when we check if the current assumptions for x validate local assumption for x, we are permitted to instantiate b to any index object i. In this example, we could substitute a for b. As a result, we end up checking $(\lambda z.\, x) \downarrow \mathbf{1} \to \mathsf{list}(a)$, even though the annotation does not mention a. Note that in an annotation $e : (\Gamma_0 \vdash A_0), As$, all index variables declared in Γ_0 are considered bound and can be renamed consistently in Γ_0 and A_0. In contrast, the free term variables in Γ_0 may actually occur in e and so cannot be renamed freely.

```
true ≼ bool,   false ≼ bool

even ≼ nat,   odd ≼ nat

evenlist ≼ list,   oddlist ≼ list,
  emptylist ≼ evenlist,   emptylist ≼ oddlist

eq : (even * odd → false)
   ∧ (odd * even → false)
   ∧ (nat * nat → bool)

(* member :  ⊢ (even * oddlist → false)
               ∧ (odd * evenlist → false)
               ∧ (nat * list → bool)  *)
fun member (x, xs) =
  (* mem : x:even ⊢ (evenlist → bool) ∧ (oddlist → false),
           x:odd ⊢ (evenlist → false) ∧ (oddlist → bool),
           x:nat ⊢ natlist → bool  *)
  let fun mem xs =
    case xs of Nil ⇒ false
     | Cons(y, ys) ⇒ eq(x, y) orelse mem ys
  in mem xs end

(* append : Πa:𝒩. Πb:𝒩. list(a) * list(b) → list(a + b)  *)
fun append (xs, ys) =
  (* app : (c:𝒩, ys:list(c) ⊢ Πa:𝒩. list(a) → list(a + c))  *)
  let fun app xs = case xs of Nil ⇒ ys
     | Cons(x, xs) ⇒ Cons(x, app xs)
  in app xs end
```

Figure 4: Example of contextual annotations

I give two small examples (from [DP04]) in Figure 4. [DP04] includes a more thorough discussion of contextual typing annotations, gives inference rules for contextual subtyping, and proves a completeness result [DP04, Corollary 12] stating that annotations can be added to any term that is well typed in the type assignment system to yield a well typed term in the tridirectional system.

4.8 Properties of typing

If we take the type system just described, replace all $\uparrow$ and $\downarrow$ with :, and remove the rule for annotations, we get the (undecidable) type assignment system of [DP03]. Not surprisingly, we also get an erasure result. Let $|e|$ denote the erasure of all typing annotations from e.

Theorem 1 (Soundness, Tridirectional). *If* $\Gamma \vdash e \uparrow A$ *or* $\Gamma \vdash e \downarrow A$ *then* $\Gamma \vdash |e| : A$.

In [DP03] we proved Theorem 2.[6]

Theorem 2 (Type Preservation and Progress in the Type Assignment System). *If* $\vdash e : A$ *then either*

[6] The theorem in the paper involves a substitution; taking the substitution to be empty yields the specialized formulation given here.

(1) e value*, or*

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

4.9 The left tridirectional system

In the tridirectional system described, the contextual rules are highly nondeterministic. Not only must we choose which contextual rule to apply, but each rule can be applied repeatedly with the same context E; for (direct), which does not even break down the type of e', this repeated application is quite pointless. Hence, we define a *left tridirectional system* system with only one contextual rule that disallows repeated application. Inspired by the sequent calculus formulation of Barbanera et al. [BDCd95], it replaces the contextual rules with (1) a slightly altered version of (direct) and (2) several *left rules* such as ($\vee\mathbb{L}$) below, where Δ is a *linear context* of variables $\overline{x}, \overline{y}, \ldots$ that must appear exactly once and in evaluation position in e.

$$\frac{\Gamma; \Delta, \overline{x}{:}A \vdash e \downarrow C \qquad \Gamma; \Delta, \overline{x}{:}B \vdash e \downarrow C}{\Gamma; \Delta, \overline{x}{:}A \vee B \vdash e \downarrow C} \; (\vee\mathbb{L})$$

In combination, these rules subsume the previous set of contextual rules: The left tridirectional system is sound and complete with respect to the tridirectional system discussed earlier. Moreover, a relatively simple argument suffices to show that the left tridirectional system is decidable. These results are proved in [DP04].

5 Constraint solving

Integer constraint solving, while NP-complete, is a well studied problem for which practical algorithms such as Fourier-Motzkin [DE73] and the Omega test [Pug91] have been developed. Xi successfully used a variant of Fourier-Motzkin in his implementation of DML.

In order to move beyond the domain of integers, it should be as easy as possible for the user to develop and use constraint solvers for new domains. A basic requirement is a clean interface allowing users to plug in one or more constraint solvers. Furthermore, the solvers must be able to work together in situations such as the following, in which a, b, c are integers and ℓ_1, ℓ_2 are lists, with one constraint solver for each domain (integers and lists):

$$a \leq b, \;\; b \leq c, \;\; \ell_1 \doteq \mathsf{Cons}(a-1, \ell_2) \;\; \models \;\; \mathsf{head}(\mathsf{append}(\ell_1, \ell_2)) < c$$

The integer constraint solver alone cannot show the satisfaction relation holds, because it cannot deduce $\mathsf{head}(\mathsf{append}(\ell_1, \ell_2)) = \mathsf{head}(\ell_1) = a - 1$. Likewise, the list constraint solver cannot deduce $a - 1 < c$. There are several methods for building such *cooperating decision procedures*; the relatively straightforward method of Nelson and Oppen [NO79] should suffice for initial experiments, and there is an extensive literature to draw upon for other techniques [Sho84, RS01, SR02].

So far so good; but we can go further. The techniques needed in different index domains often have much in common. For example, a term rewriting engine (where the terms are index expressions) might be useful for both a domain of lists (where the operations are cons, head, tail, and append) and a domain of format strings (in the style of C's 'printf'). Hence, a constraint solver consisting of a term rewriting engine, which one would parameterize by particular sets of rewrite rules, would be desirable.

There are already solvers that apply cooperating decision procedures, such as CVC [SBD02]. It might be expedient to use one of these systems instead of building my own.

6 Reaching beyond the envelope

The central theme of my work is to expand the capabilities of static type systems (the "envelope") by making more properties expressible. Yet there will always be classes of properties

that cannot be expressed without losing decidability. Furthermore, components such as object-code libraries are simply impervious to typechecking. While these difficulties are fundamental to our approach, I suggest several ways of mitigating them.

The first is a fallback strategy of *runtime refinement checking*. For example, if we claim that some library function f has type

$$f : \Pi a{:}\mathcal{N}.\, \mathsf{int}(a) \to \mathsf{list}(a)$$

we could wrap every call to f with a check that the length of the resulting list is equal to its argument. One mechanism for implementing this would be a *soft annotation*. The idea is that we erase the annotation if the annotated term can be checked against the type; if it cannot (for example, if it is defined in an external library), we generate a wrapper function that includes a call to an automatically generated checking function.[7]

As the above example illustrates, such checks could be linear in the size of the data structures involved. Moreover, if the data is functional, we would need to build a new copy of the data with the functions replaced by wrapped functions. This suggests a use for a third kind of annotation, the *sharp annotation*, which is *not even checked at runtime*. This is quite dangerous (hence the name): if the property claimed by the sharp annotation does *not* hold, the typechecker will happily reason from that false assumption, sowing confusion.

A more compelling need for sharp annotations arises out of the index refinements. Some constraint domains involve properties that are *not even checkable at runtime*. A simple example is found in the seemingly innocuous theory of order, where the objects of discourse have an operation $\preceq$ which must be reflexive, antisymmetric and transitive. If we use this theory with a primitive type such as the integers and assume the built-in integer comparison operator has the necessary properties, all is well. But if we need the theory of order for some user-defined type, such as keys in a database with various components lexicographically ordered, the user writes some comparison function *cmp*, so we should show that *cmp* is reflexive, antisymmetric and transitive. However, the property of transitivity is neither expressible in the refinement system nor checkable at runtime! In this case, the form of the sharp annotation is different:

$$cmp \text{ :\#: TRANSITIVE}$$

The constraint solver furnishes the information that TRANSITIVE denotes a property. There is no check that *cmp* actually is transitive, but at least the programmer is forced to explicitly claim the property holds.

Ideally, the system would be able to use evidence of various kinds to support sharp annotations. A key goal of the Programatica project [Pro03] is to build a system that tracks various kinds of evidence, from mere claims of the kind just discussed to proofs produced by full-blown theorem provers. Integrating type refinements with a Programatica-like system would be interesting and could be very powerful, but is beyond the scope of this work.

A final note: I discuss external libraries because they are obviously impervious to static typechecking. But programs can grow so large that parts of the program may as well be an external library! Suppose that in a large project, there is a module M with datatype τ which M's programmer refined as she saw fit; years later someone working on another part of the program needs an unforeseen refinement of τ, but the benefit from static checking may not justify the effort involved in modifying M, especially if the new refinement requires even minor adjustments to the algorithms. In such a situation, soft or sharp annotations would be useful.

7 Related work

In this section, I briefly examine a few other approaches to the problem of verifying software properties, and compare them to type refinements.

[7] Of course, the programmer could write the checking functions by hand, but the process should be automated.

7.1 Assertions

Assertion mechanisms allow any boolean expression to be tested at runtime. Such assertions serve as documentation and also encourage "early failure", making bugs easier to pinpoint. Of course, the lack of restrictions on the form of the asserted expression makes compile-time assertion checking undecidable.

We can view a function type $A \to B$ as an implicit pair of assertions: a precondition of the form "the argument is of type A" and a postcondition of the form "the result is of type B". With type refinements, pre- and postconditions of this form are more expressive than in a conventional static type system. Thus, type refinements reduce but do not eliminate the utility of generic assertion facilities such as those of Findler and Felleisen [FF02].

Note that under certain circumstances, a compiler can optimize away the assertion check: if dataflow analysis shows that $x = 1$ when $\mathit{assert}(x > 0)$ is reached, the assertion is equivalent to $\mathit{assert}(\texttt{true})$ which is a no-op. However, most compilers provide little feedback as to which checks are actually removed, so this is not much help.

7.2 ESC

ESC, the Extended Static Checker [DLNS98, Lei01] is intended to help find bugs in Modula-3 and Java programs. Annotations can be written expressing bounds on integers and synchronization properties, among others. ESC then checks the program in light of the annotations, and reports potential problems. These are only potential problems: for example, ESC may report a potential null pointer dereference when in fact the pointer can never be null. Furthermore, ESC is *unsound*: it may incorrectly report that there are no problems. In practice, though, the system has been good at finding bugs.

Type refinements are not as expressive as the property language of ESC. However, even if we restrict our attention to a subset of the ESC properties that can be expressed through type refinements—for example, a pointer being non-null corresponds to a value of `option` type having a particular datasort refinement—ESC can get away with having less information (annotations), since it uses several tricks to (unsoundly) finesse undecidable problems such as generating loop invariants. The designers of ESC argue that much can be gained by giving up the "shackles of soundness" [DLNS98, p. 33], but the power and ease of use of the individual type refinement systems demonstrates that we have only begun to see how far we can go *without* giving up soundness. It is also interesting to note that Leino [Lei01], in his discussion of "future challenges", suggests exploring "more-than-types systems"—by which he means approximately what I mean by "type refinements", though he focuses on imperative languages rather than functional languages. Finally, ESC has **assume** statements roughly corresponding to my "sharp annotations", and Leino suggests falling back on dynamic checks (my "soft annotations") or **assume** statements in cases where static checking is undecidable.

7.3 Testing

Refined type systems will never be powerful enough to make testing completely unnecessary, but we could facilitate the automatic generation of test inputs by excluding inputs that are well typed in a conventional type system, but ill typed in the refined type system. However, unless the refinements *exactly* describe the property of being a valid input, additional programmer guidance is needed to generate the inputs.

7.4 Soft typing

Traditionally, dynamically typed languages require many so-called *runtime type checks*. *Soft typing* systems [CF91, AWL94] aim to reduce the number of runtime type checks by analyzing the

program, yielding constraints on data values, allowing some checks to be removed. For example, in the program

```
(car '(1 2 3))
```

the primitive *car*, which returns the head of its argument if the argument is a list, need not check that its argument really is a list: it is clear from the analysis that the only value ever passed to it (at this program point) is a list, namely `'(1 2 3)`.

If we view such type systems as *unityped* systems where there is one type, which happens to be an algebraic datatype with alternatives corresponding to cons cells, integers, functions, etc., these "runtime type checks" are clearly *runtime tag checks*.

$$\begin{aligned}\textbf{datatype}\ \mathsf{dynamic} = {} & \mathsf{Cons}\ \textbf{of}\ \mathsf{dynamic} * \mathsf{dynamic} \\ & \mid \mathsf{Int}\ \textbf{of}\ \mathsf{int} \\ & \mid \mathsf{Func}\ \textbf{of}\ \mathsf{dynamic} \rightarrow \mathsf{dynamic} \\ & \vdots\end{aligned}$$

Through datasort refinements, we can express the property that a value of some datatype is, in particular, some specific constructor. For instance, we can declare a datasort intsort of integers and a datasort intfunc of functions from integers to integers.

$$\begin{aligned}\textbf{datasort}\ \mathsf{intsort} &= \mathsf{Int}\ \textbf{of}\ \mathsf{int} \\ \textbf{and}\ \mathsf{intfunc} &= \mathsf{Func}\ \textbf{of}\ \mathsf{intsort} \rightarrow \mathsf{intsort}\end{aligned}$$

Unfortunately, these datasort definitions do not strictly fit the framework of regular tree grammars: the definition of intfunc includes an arrow, $\mathsf{intsort} \rightarrow \mathsf{dynamic}$, and in particular a negative occurrence of a datasort (intsort on the left of the arrow). Davies has extended standard algorithms on regular tree grammars to handle datasorts in negative positions, but he has not proved formal properties of the extension, and experience with the extension is limited [Dav03]. Thus, we cannot yet conclude that datasort refinements subsume soft typing.

7.5 Cayenne: unrestricted dependent typing

Index refinements can be seen as a restricted form of dependent types. Cayenne [Aug98] is an extension of Haskell with *unrestricted* dependent types. Consequently, typechecking in Cayenne is undecidable; the Cayenne typechecker "times out" if typechecking takes more than a specified number of steps. Cayenne does not define any kind of datatype refinement; its main concern is with admitting *more* programs (such as a typechecked analogue of C's `printf`) rather than fewer. It is not obvious if the Cayenne approach could be extended with datatype refinements in a sensible way.

7.6 Phantom types

The technique known as *phantom types* can express, through a clever use of parametric polymorphism and modularity, some of the properties expressible through type refinements. The idea is to "tack on" an extra type variable, to be instantiated with an "index" in the form of a type. Take a type atom and a function *conj*:

$$\textbf{datatype}\ \mathsf{atom} = \mathsf{Int}\ \textbf{of}\ \mathsf{int} \mid \mathsf{Bool}\ \textbf{of}\ \mathsf{bool}$$

$$\begin{aligned}\textbf{fun}\ \textit{conj}\ \ (\mathsf{Bool}\ \mathtt{p},\ \mathsf{Bool}\ \mathtt{q}) &= \mathsf{Bool}(\mathtt{p}\ \textbf{andalso}\ \mathtt{q}) \\ \mid (_,_) &= \textbf{raise}\ \textit{Error}\end{aligned}$$

Declare a new type expr, "index" types intty, boolty, and wrappers around the constructors of atom.

$$\begin{aligned}
&\textbf{datatype}\ \alpha\ \mathsf{expr} = \mathsf{Expr}\ \textbf{of}\ \mathsf{atom}\\
&\textbf{type}\ \mathsf{intty}\\
&\textbf{type}\ \mathsf{boolty}\\
&\textbf{fun}\ \mathit{mkInt}\ \mathtt{i}\ =\ \mathsf{Expr}(\mathsf{Int}(\mathtt{i})) : \mathsf{intty\ expr}\\
&\textbf{fun}\ \mathit{mkBool}\ \mathtt{p}\ =\ \mathsf{Expr}(\mathsf{Bool}(\mathtt{p})) : \mathsf{boolty\ expr}
\end{aligned}$$

Note that intty and boolty are kept opaque—their actual definition is irrelevant. The annotations on the bodies of *mkInt* and *mkBool* constrain types that would otherwise be polymorphic, α expr. Now modify *conj* thus:[8]

$$\begin{aligned}
&\mathit{conj}\ :\ \mathsf{boolty\ expr} * \mathsf{boolty\ expr} \to \mathsf{boolty\ expr}\\
&\textbf{fun}\ \mathit{conj}\ \ (\mathsf{Expr}(\mathsf{Bool}\ \mathtt{p}),\ \mathsf{Expr}(\mathsf{Bool}\ \mathtt{q}))\ =\ \mathit{mkBool}(\mathtt{p}\ \textbf{andalso}\ \mathtt{q})\\
&\qquad\quad \mid (_,_)\ =\ \textbf{raise}\ \mathit{Error}
\end{aligned}$$

Then ordinary Hindley-Milner typing suffices to reject expressions such as

$$\mathit{conj}(\mathit{mkBool}(\mathsf{true}), \mathit{mkInt}(42))$$

There are several drawbacks to the phantom types approach. First, it is (at least to the uninitiated) quite unnatural; the above example is rather straightforward but more complicated uses of phantom types, such as modeling array length [Blu01], are not. Second, it does not yield any inversion properties: assuming v : boolty expr, we cannot show v is an instance of the constructor Bool, and therefore cannot prove that the **raise** *Error* arm in *conj* is unreachable. The latter drawback appears to be addressed by Cheney and Hinze [CH03], but to do so they must build phantom types into the type system, eliminating a key advantage of phantom types: compatibility with conventional type systems.

Fluet and Pucella [FP02] use phantom types to model certain forms of subtyping, and cite numerous other applications of phantom types. The example was adapted from Leijen and Meijer [LM99].

8 Road map

I have already completed substantial work towards my thesis:

- Combined datasort and index refinements into a single type system with intersection types and universal index quantification [Dun02].
- Formulated a clean (but undecidable) type assignment system with datasort and index refinements, intersections, unions, universal and existential index quantification, and least and greatest types [DP03].
- Formulated a decidable version of the type assignment system, via bidirectional typing and *typing annotations* [DP04].

To complete the thesis, I plan to do the following:

1. Formulate a let-normal transformation and associated variant of the decidable bidirectional system such that the let-normal system is sound and complete with respect to the earlier system, yet substantially less nondeterministic.

[8] The annotation syntax is not valid ML.

2. Elaborate the present type system, which assumes the typechecker can guess index expressions when instantiating index quantifiers, into a constraint-generating system.

3. To facilitate experimentation, adapt the type system to a small functional language (somewhere between the system of [DP04] and the core language of SML) and implement it. Initially I will use "hardcoded" refinements, that is, the user explicitly gives the refined types of data constructors.

4. Apply the system to a variety of problems. I particularly want to explore a wide range of index domains. Some of these domains should fit the current framework well, for example:

   - Sets, ordered sets, multisets; there are some nice decidable fragments of these theories, such as the monadic theory of order [She75].
   - Dimension types [Ken94].

   Other domains would likely require modification to the type system:

   - Terms in the Twelf language (Sarkar and Crary).
   - Termination checking [Xi01].
   - Effects (properties like those considered by Mandelbaum et al. [MWH03]).

5. Identify and address performance bottlenecks (such as, perhaps, left rule nondeterminism).

6. Write the dissertation.

References

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *Int'l Conf. Functional Programming*, pages 239–250, 1998.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symp. Principles of Programming Languages*, pages 163–173, 1994.

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.

[Blu01] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[Cam] Caml Light website. `http://caml.inria.fr`.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 1997. Release of 1 October 2002.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[Dav97] Rowan Davies. A practical refinement-type checker for Standard ML. In *Algebraic Methodology and Software Tech. (AMAST'97)*, pages 565–566. Springer LNCS 1349, 1997.

[Dav03] Rowan Davies. Personal communication, November 2003.

[Dav04] Rowan Davies. *Practical refinement-type checking*. PhD thesis, Carnegie Mellon University, 2004. To appear.

[DE73] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *J. Combinatorial Theory (A)*, 14:288–297, 1973.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-159, Compaq SRC, Palo Alto, CA, 1998.

[DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000.

[DP03] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computational Structures (FOSSACS '03)*, pages 250–266, Warsaw, Poland, April 2003. Springer LNCS 2620.

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In Xavier Leroy, editor, *ACM Symp. Principles of Programming Languages (POPL '04)*, Venice, Italy, January 2004. To appear.

[Dun02] Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Int'l Conf. on Functional Programming (ICFP'02)*, pages 48–59, October 2002.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 268–277. ACM Press, 1991.

[FP02] Matthew Fluet and Ricardo Pucella. Phantom types and subtyping. In *2nd IFIP Int'l Conf. Theoretical Computer Science (TCS '02)*, pages 448–460, August 2002.

[Fre94] Tim Freeman. *Refinement types for ML*. PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-110.

[HP99] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[Ken94] Andrew Kennedy. Dimension types. In *European Symposium on Programming (ESOP '94)*, volume 788, pages 348–362. Springer, 1994.

[Lei01] K. Rustan M. Leino. Extended Static Checking: A ten-year perspective. *LNCS*, 2000:157–175, 2001.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Domain-Specific Languages*, pages 109–122, 1999.

[ML] ML Kit website. `http://www.itu.dk/research/mlkit/`.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Int'l Conf. Functional Programming (ICFP '03)*, pages 213–226, Uppsala, Sweden, September 2003.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Sys.*, 1(2):245–257, 1979.

[Pie91a] Benjamin C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-205.

[Pie91b] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[Pro03] Programatica website. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`, 2003.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *ACM Symp. Principles of Programming Languages*, pages 252–265, 1998. Full version in *ACM Trans. Prog. Lang. Sys.*, 22(1):1–44, 2000.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conf. Supercomputing*, pages 4–13, 1991.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th IEEE Symp. Logic in Computer Science (LICS '01)*, pages 19–28, 2001.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *14th Int'l Conf. Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.

[She75] Saharon Shelah. The monadic theory of order. *Annals of Mathematics*, 102:379–419, 1975.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *Int'l Conf. Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.

[Xi98] Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 231–242, June 2001.

[Xi02] Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, October 2002.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symp. Principles of Programming Languages*, pages 214–227, 1999.