Annotations for Intersection Typechecking

Joshua Dunfield
Max Planck Institute for Software Systems
Kaiserslautern and Saarbrücken, Germany
joshua@mpi-sws.org

In functional programming languages, the classic form of annotation is a single type constraint on a term. Intersection types add complications: a single term may have to be checked several times against different types, in different contexts, requiring annotation with several types. Moreover, it is useful (in some systems, necessary) to indicate the context in which each such type is to be used.

This paper explores the technical design space of annotations in systems with intersection types. Earlier work (Dunfield and Pfenning 2004) introduced *contextual typing annotations*, which we now tease apart into more elementary mechanisms: a "right hand" annotation (the standard form), a "left hand" annotation (the context in which a right-hand annotation is to be used), a *merge* that allows for multiple annotations, and an existential binder for index variables. The most novel element is the left-hand annotation, which guards terms (and right-hand annotations) with a judgment that must follow from the current context.

1 Introduction

The origin of intersection lay in the analysis of the solvability of λ-terms; the key early result was that, in a system with $\to$ and $\wedge$, typeability and strong normalization coincide (Coppo et al. 1981). While pure type assignment is thus undecidable for intersection types, systems that *check* types of lightly-annotated programs, including systems based on bidirectional typechecking, have had some success. But constructing a type-checking system from a type assignment system is not trivial. A key issue is the design of the annotations. The classic annotation form $(e : A)$, which merely marks a term with a single type, fails in intersection type systems that must check the same term several times, in different contexts. Furthermore, in systems with indexed types, we run into problems with the scope of index variables; the simple mechanism of a term-level binder fails, because intersections can be formed from types with different numbers of quantifiers.

For guidance, we can look to logic and the form of hypothetical judgments: in $\Gamma \vdash \Delta$ we have, on the left, assumptions Γ (implicitly conjoined, because we wish to make several assumptions, each definite); on the right, we have conclusion Δ. In the sequent calculus (Gentzen 1969), the conclusion is plural and implicitly *dis*joined: from a conjunction of assumptions, we conclude a disjunction of conclusions. This conforms to the internal duality of the sequent calculus.

The classic annotation form, $e : A$, seems to be "on the right". It is an obligation that constrains the type of e: "I insist that e have type A, and if you cannot satisfy this demand, typechecking should fail." (The term e might have some other type B, but unless B is a subtype of A the demand is not met. Also, in typecheckers that backtrack, like the intersection-type checkers considered in this paper, the requirement that "typechecking should fail" means that the particular typing subproblem fails—the program could still typecheck.) Writing $(e : A)$ does not correspond to having an assumption $e : A$, because that would let us assume that e has type A, even if it should not have that type. Further evidence in support of right-handedness is that several systems with intersection types allow lists of types in annotations, and

S. Graham-Lengrand and L. Paolini (Eds.): Sixth Workshop on
Intersection Types and Related Systems (ITRS 2012).
EPTCS 121, 2013, pp. 35–47, doi:10.4204/EPTCS.121.3

these lists are interpreted disjunctively, consistent with the sequent calculus where lists of conclusions are interpreted disjunctively.

If the classic annotation $(e : A)$ is "on the right", what form of annotation is "on the left"? It is hard to imagine an annotation that is not an obligation, or does not contribute to an obligation (leaving aside the sort of annotation that is an explicit direction to ignore truth and charge ahead, as with the `admit` of Coq (Coquand et al. 2012) or the `%trustme` of Twelf (Pfenning et al. 2012)).

We can, however, distinguish annotations that carry an obligation with respect to the term on the right of the turnstile, such as $(e : A)$, from those that carry an obligation with respect to the assumptions on the *left* of the turnstile. Writing such a "left-hand" annotation says, "I insist on something about the assumptions you have when you type this term, and if you cannot satisfy me, give up." Since the point of an assumption is to help conclude things, the "something about the assumptions" should be about what those assumptions entail. The most direct entailment is the use of a hypothesis: if $\Gamma = \{\Gamma_1, \ldots, \Gamma_n\}$ then $\Gamma \vdash \Gamma_k$ for $1 \leq k \leq n$, suggesting that we should be able to write part of a context as a left-hand annotation.

The last piece of the puzzle is a way of writing more than one (right-hand) annotation. It suffices to support a well-behaved special case of the unruly *merge construct* (Dunfield 2012).

Contents We start by giving an overview of annotations in intersection type systems (Section 2), then describe a language whose most notable features are the left-hand *guard annotation* (Section 3) and a merge construct (Section 4). Next, we extend that language with indexed types (Section 5); the presence of index variables leads us to another construct (an existential binder for index variables). In Section 6, we show that the features of the extended language—left- and right-hand annotations, plus the merge construct and the existential binder—collectively subsume the *contextual typing annotations* developed in earlier work (Dunfield and Pfenning 2004), replacing one complicated construct with several simpler ones. Section 7 compares our approach to contextual modal types. Finally, we briefly discuss a prototype implementation (Section 8) and speculate on the usability of the approach (Section 9).

2 Overview

For languages based on the ordinary λ-calculus, the usual form of annotation is a single type, either around a term $(e : A)$ or on a bound variable $(\lambda x : A.\, e)$. In such languages, the single type corresponds to typing: exactly one subderivation types each subterm e.

In languages with intersection types, the introduction rule for intersection replicates the same term in each premise:

$$\frac{\begin{array}{cc}\mathscr{D}_1 & \mathscr{D}_2 \\ \Gamma \vdash e : A_1 & \Gamma \vdash e : A_2\end{array}}{\Gamma \vdash e : A_1 \wedge A_2}\ \wedge\mathrm{I}$$

Both $\mathscr{D}_1$ and $\mathscr{D}_2$ have as conclusion a typing for e; in general, neither A_k is a subtype of the other. In general, we need both derivations because the differences between A_1 and A_2 can lead to structural differences in $\mathscr{D}_1$ and $\mathscr{D}_2$, and even in the contexts used inside $\mathscr{D}_1$ and $\mathscr{D}_2$.

Assume a subtyping system in which the type bits of bitstrings is refined by odd and even, denoting bitstrings of odd and even parity (having an odd or even number of 1s). Appending a 1 (written $x \cdot 1$) should flip the parity, so

$$(\lambda x.\, x \cdot 1) \ :\ (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$$

In the typing derivation, we assume $x : \mathsf{odd}$ inside the first branch of $\wedge$I and $x : \mathsf{even}$ inside the second:

$$\dfrac{\dfrac{x : \mathsf{odd} \vdash x \cdot 1 : \mathsf{even}}{\cdot \vdash (\lambda x.\, x \cdot 1) : (\mathsf{odd} \to \mathsf{even})} \qquad \dfrac{x : \mathsf{even} \vdash x \cdot 1 : \mathsf{odd}}{\cdot \vdash (\lambda x.\, x \cdot 1) : (\mathsf{even} \to \mathsf{odd})}}{\cdot \vdash (\lambda x.\, x \cdot 1) \;:\; (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})} \wedge\text{I}$$

This function $\lambda x.\, x \cdot 1$ is very simple; assuming the goal type $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$ is already known, any reasonable typechecker should handle it without annotations inside the function body. But more complicated code might require internal annotation. Anyway, programmers should be able to write unnecessary annotations if they want to.

Here, there is no single type we can write for the use of x in $x \cdot 1$: in the left side of the derivation, x has type odd, and in the right side, x has type even. To handle this issue, several systems with intersection types allow *lists* of types in annotations: Forsythe (Reynolds 1988, 1996) and Pierce (1991, p. 21) allow λ arguments to be annotated with a sequence of types: $\lambda x : \mathsf{odd}|\mathsf{even}.\, x \cdot 1$; the refinement typechecker SML-CIDRE (Davies 2005) allows terms to be annotated with lists of types, so we could write $\lambda x.\, (x : \mathsf{odd}, \mathsf{even}) \cdot 1$.

Intersection type inference is undecidable, but even intersection type *checking* is PSPACE-hard. Unfortunately, unlike Hindley-Milner inference, which is intractable in theory but polynomial in practice, intersection typechecking is expensive in practice (Dunfield 2007a). A system should, therefore, give the user a rich set of tools—such as annotations—to help make typechecking practical.

Finally, in systems with indexed types and index-level variables, we need to resolve a conflict between orderly variable scoping and intersection types.

Earlier work (Dunfield and Pfenning 2004) described a *contextual typing annotation* that combined several features:

- *contextuality*, guarding the type in the annotation with the context in which it makes sense;
- *multiplicity*, allowing more than one typing to be given, corresponding to different branches of intersection;
- *index variable linking*, maintaining index variable scoping even with intersection types.

We now recast the contextual typing annotation, separating it into constituent mechanisms that collectively subsume it. For contextuality, we introduce a *guard* construct. For multiplicity, we use a *merge construct* (Dunfield 2012). For index variable linking, we propose an existential binder.

3 A Language with Guard Annotations

We'll use a small functional language with intersection types, a merge construct, and two kinds of annotations (Figure 1).

3.1 Bidirectional Typechecking

Our type system is *bidirectional* (Pierce and Turner 2000; Dunfield and Pfenning 2004; Dunfield 2009); see Dunfield (2009) for background. This technique offers two major benefits over Damas-Milner type inference: it works when annotation-free inference is undecidable, and it produces more localized error messages. Unlike constraint-based type inference, bidirectional typechecking does not inherently require

Types	A, B, C	$::= \mathsf{unit} \mid A \to B \mid A \wedge B$	
Terms	e	$::= x \mid () \mid \lambda x.\, e \mid e_1\, e_2$	
		$\mid (e : A)$	standard ("right-hand") annotation
		$\mid d :\!> e$	guard ("left-hand") annotation
		$\mid e_1 \,„\, e_2$	merge
Declarations	d	$::= x : A$	
Contexts	Γ	$::= \cdot \mid \Gamma, d$	

Figure 1: Types, terms, declarations and contexts

unification, nor the generation or manipulation of any constraints. The basic idea of bidirectional typechecking is to separate checking of a term against a known type from synthesis of an unknown type: $\Gamma \vdash e \Leftarrow A$ means that e checks against known type A, while $\Gamma \vdash e \Rightarrow A$ means that e synthesizes type A. In the checking judgment, Γ, e and A are inputs to the typing algorithm. In the synthesis judgment, Γ and e are inputs and A is output. As usual, declarations of the form $x : A$ are added to Γ through $\to$-introduction (rule $\to$I); unlike in the Damas-Milner framework, the type added is not a unification variable but a closed type. In $\to$I, the type A comes from the type $A \to B$ that the λ-expression is checked against.

Bidirectional typechecking does need more type annotations than type inference. However, by following the approach of Dunfield and Pfenning (2004)—checking introduction forms (like $\lambda x.\, e$) and synthesizing the types of elimination forms (like $e_1\, e_2$)—annotations are required only on redexes like $(\lambda x.\, e_1)e_2$ and recursive function declarations. The need for annotations is thus predictable; variations and refinements of this basic approach (such as trying to synthesize the types of introduction forms) can further reduce the volume of annotations.

While we omit parametric polymorphism from this paper to focus on issues specific to intersection types, it is straightforward to support parametric polymorphism, *if* type abstraction and application are explicit: given $e : \forall \alpha.\, B$, write $e[A]$ to instantiate α at A. Such explicit instantiation is very inconvenient for the programmer. It is possible, but not entirely straightforward, to extend bidirectional typechecking with a form of existential type variable Dunfield (2009). This algorithm removes the need for explicit instantiation, yet does not use unification, relying instead on a form of matching.

3.2 Merging

If either e_1 or e_2 has type A, then the merge $e_1 \,„\, e_2$ has type A. This construct first appeared in Forsythe (Reynolds 1996). Used in full generality (Dunfield 2012), the merge can encode a variety of type system features, requires an elaboration-based semantics, and leads to ambiguity if e_1 and e_2 have different operational behaviour. In the present setting, the purpose of the merge is just to let us annotate the same term in different ways. Used in this restricted fashion, erasing annotations from e_1 and e_2 yields the same term; thus, e_1 and e_2 have the same operational behaviour. We discuss this point further in Section 4.

Since the merge is neither an introduction nor an elimination form, we can give a synthesizing rule in addition to a checking rule; see Figure 2.

Using a merge, the example $\lambda x.\, x \cdot 1$ from the introduction can be annotated as follows:

$$\lambda x.\ (x \cdot 1 : \mathsf{even}) \,„\, (x \cdot 1 : \mathsf{odd})$$

Subtyping

$$\frac{}{\Gamma \vdash A \leq A}\ \text{refl}\leq \qquad \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \to\leq$$

$$\frac{\Gamma \vdash A_k \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ \wedge L_k\leq \qquad \frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ \wedge R\leq$$

Variables, unit, $\to$

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x \Rightarrow A}\ \text{var} \qquad \frac{}{\Gamma \vdash () \Leftarrow \mathsf{unit}}\ \text{unitI}$$

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

Intersection, subsumption, merge

$$\frac{\Gamma \vdash e \Leftarrow A_1 \qquad \Gamma \vdash e \Leftarrow A_2}{\Gamma \vdash e \Leftarrow A_1 \wedge A_2}\ \wedge I \qquad \frac{\Gamma \vdash e \Rightarrow A_1 \wedge A_2}{\Gamma \vdash e \Rightarrow A_k}\ \wedge E_k$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e \Leftarrow B}\ \text{sub} \qquad \frac{\Gamma \vdash e_k \Leftarrow A}{\Gamma \vdash e_1 \,„\, e_2 \Leftarrow A}\ \text{merge}\Leftarrow_k \qquad \frac{\Gamma \vdash e_k \Rightarrow A}{\Gamma \vdash e_1 \,„\, e_2 \Rightarrow A}\ \text{merge}\Rightarrow_k$$

Annotations

$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}\ \text{right-anno}$$

$$\frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash e \Leftarrow B}{\Gamma \vdash x : A >:> e \Leftarrow B}\ \text{left-anno}\Leftarrow \qquad \frac{\Gamma \vdash x \Leftarrow A \qquad \Gamma \vdash e \Rightarrow B}{\Gamma \vdash x : A >:> e \Rightarrow B}\ \text{left-anno}\Rightarrow$$

Figure 2: Subtyping and typing rules

so it checks against $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$.

3.3 Guard Annotations

Checking a function against intersection type leads to the function body being checked several times against different return types, and even under varying typings of the function's argument. The latter motivates *guards*. A guard $d >:> e$ protects a term e (say, the body of a function) with a declaration, so that the current typing context Γ must support the guarding declaration d. For variable declarations $x : A$, this amounts to $\Gamma \vdash x \Leftarrow A$.

We have both synthesis and checking typing rules for guards, ensuring that guards can be placed anywhere the user chooses.

Using guards, we can annotate the example $\lambda x.\, x \cdot 1$ so that the choice of branch is fully determined:

$$\lambda x.\ \big(x : \mathsf{odd} >:> (x \cdot 1 : \mathsf{even})\big) \,„\, \big(x : \mathsf{even} >:> (x \cdot 1 : \mathsf{odd})\big)$$

3.4 Free Annotation

Given a term e that can be typed with the bidirectional rules—that is, a term that already has enough annotations for the typechecker—the user can freely choose to put in more annotations, either right-hand annotations or guards. If different annotations are needed in the subderivations of $\wedge$I, the user can duplicate the term with a merge.

4 Operational Semantics of Annotations and Merges

We are working with a bidirectional type system. For such a system, the easiest way to translate the usual notions of preservation and progress is to give an equivalent type assignment system. That is, we want rules deriving $\Gamma \vdash e : A$ such that:

(1) if $\Gamma \vdash e \Leftarrow A$, then $\Gamma \vdash e : A$;

(2) if $\Gamma \vdash e \Rightarrow A$, then $\Gamma \vdash e : A$.

To show *equivalence*, we would also need to consider the other direction: given some $\Gamma \vdash e : A$, can we derive appropriate bidirectional judgments? We need not answer this question to describe the operational semantics; see Dunfield and Pfenning (2004) for one answer.

4.1 Left- and Right-Hand Annotations

For standard and guard annotations, we can give a small-step operational semantics, but we have a choice of approaches. The first approach—standard in typed functional languages—is to erase the annotations, so that the operational semantics does not mention them at all. In this approach, we define an erasure function $|e|$:

$$\begin{array}{rcl} |x| & = & x \\ |()| & = & () \\ |\lambda x.\, e| & = & \lambda x.\, |e| \\ |e_1\, e_2| & = & |e_1|\,|e_2| \\ |(e : A)| & = & |e| \\ |d :> e| & = & |e| \end{array}$$

The typing rules for left- and right-hand annotations have premises typing the inner expression e, so this erasure function clearly preserves types. Since $(e : A)$ and $d :> e$ get erased, they need no reduction rules.

The second approach is to extend the definition of values:

$$v \quad ::= \quad x \mid \lambda x.\, e \mid (v : A) \mid d :> v$$

and give reduction rules that drop the annotations.

$$\frac{}{(e : A) \;\mapsto\; e} \qquad\qquad \frac{}{d :> e \;\mapsto\; e}$$

As just noted, the typing rules for these constructs have premises typing e, so we can readily extend an existing proof of type preservation to handle these new reduction rules. Moreover, progress is maintained: if $(e : A)$ is well-typed and not a value, then e is not a value, and we can use the induction hypothesis on the premise typing e to show that e, which $(e : A)$ steps to, is well-typed.

4.2 Merges

We still have to deal with the merge construct. In this paper, we are interested in merges only as an annotation mechanism; merges $e_1 \,„\, e_2$ used for that purpose must have similar branches e_1 and e_2. That is, e_1 and e_2 are differently-annotated versions of some unannotated "parent" e. We can apply the first approach—erasing annotations before evaluation—by extending the definition of erasure:

$$|e_1 \,„\, e_2| \quad = \quad |e_1| \quad \text{if } |e_1| = |e_2|$$

If merges are indeed always used purely as an annotation mechanism, the side condition will always hold.

We can also try to apply the second approach of reducing annotations during evaluation, with reduction rules

$$\frac{}{e_1 \,„\, e_2 \;\mapsto\; e_1} \qquad\qquad \frac{}{e_1 \,„\, e_2 \;\mapsto\; e_2}$$

These reduction rules introduce nondeterminism. If we continue to assume, however, that e_1 and e_2 are differently-annotated versions of the same term, this nondeterminism is harmless: we will end up with the same value, no matter which rule we apply. Thus, we could omit one of the preceding two reduction rules, removing the nondeterminism.

4.3 Merges Without Restriction

Giving an operational semantics to arbitrary uses of merge, where e_1 and e_2 may be entirely different, is more involved. Dunfield (2012) gives such a semantics in two parts. The first part is a system of reduction rules, including the two above, for which the usual notions of preservation and progress fail to hold. The second part is an elaboration (more involved than erasure) to target terms M, which are evaluated by a completely standard operational semantics. This elaboration translates intersections to products (and unions to sums); the elaborating version of $\wedge\mathrm{I}$ generates a pair, and the elaborating versions of $\wedge\mathrm{E}_k$ generate projections.

The central result in that paper is that if e elaborates to M, evaluating the target term M produces a value W such that *there exists* some sequence of reductions of e that yield an equivalent value v—one such that v elaborates to W.

5 Extension to Indexed Types with Index Variables

The above constructs collectively yield annotations that work when terms are checked repeatedly under different contexts. But this does not quite subsume contextual typing annotations (Dunfield and Pfenning 2004), which were designed in the setting of a system with indexed types as well as intersection (and union) types, and treat index-level variables a, b differently from term-level variables (x, y, etc.).

After setting the stage with some background on indexed types, we look at two alternatives in language design and show how our approach works for both; for one of the alternatives, one more language construct is needed.

Index variables	a, b	
Index sorts	γ	$::= \mathsf{int} \mid \cdots$
Index expressions	i	$::= a \mid \cdots$
Index propositions	P	$::= i \doteq i \mid \cdots$
Types	A, B, C	$::= \cdots \mid \tau(i) \mid \Pi a{:}\gamma.A$
Declarations	d	$::= \cdots \mid a : \gamma$

Figure 3: Indexed types

5.1 Indexed Types

The kind of indexed types we consider here is exemplified by DML (Xi and Pfenning 1999; Xi 1998), and some of its descendants (Dunfield and Pfenning 2003, 2004; Dunfield 2007b), which added several features, most notably intersection and union types. In these systems, users can index datatypes with *index expressions* from a constraint domain with decidable equality (at least). The canonical example of such a domain is linear inequalities over integers; dimensions (metres, seconds, etc.) form another useful domain (Dunfield 2007a).

In contrast to dependent types, indices do not appear in terms e (except within annotations) and disappear completely during compilation; terms e can never appear in indices. Indexed type systems are parametric in the index domain.

We mostly follow (Figure 3) the notation of Dunfield and Pfenning (2004). Index expressions i have index sorts γ (e.g. int or dim); a and b are index-level variables standing for index expressions; P stands for propositions over index expressions, such as equality $\doteq$. Types are extended with indexed datatypes $\tau(i)$ (where τ is some inductive datatype list, tree, etc.) and universal quantification over index variables. (The use of Π is traditional and, to readers used to dependent types, has the advantage of suggesting the appropriate quantifier, with the disadvantage of being easily confused with a genuine dependent Π.) In practice, we also need existential quantification $\Sigma a{:}\gamma.A$, which we omit since it has no effect on the techniques described in this paper.

We assume that the constraint domain defines when two kinds of judgments are derivable: $\Gamma \vdash P$ (index assumptions in Γ entail index proposition P) and $\Gamma \vdash i : \gamma$ (index expression i, which might include index variables declared in Γ, has index sort γ). The only mandatory syntax in an index domain is $\doteq$, which is needed for subtyping. In practice, the index expressions i might include literal integers and operations like $i + i$; the index propositions would include comparisons like $i < i$.

Practical bidirectional typechecking with indexed types, unlike bidirectional typing for the language in previous sections of this paper, does involve constraints. However, these constraints are just over index expressions, not types, so the basic structure of the bidirectional approach need not change. For a discussion of the techniques involved, see Xi (1998) and Dunfield (2007b).

5.2 Indexed Types Without Binders

The most syntactically economical formulation of indexed types does not extend the term syntax at all (apart from the extension of the type language, which changes the syntax of annotations). Its subtyping and typing rules are shown in Figure 4. Implicitly, we assume that $\Pi R{\leq}$ and ΠI rename the variable introduced into the context if it already occurs in Γ.

Is that the end of the story? No. We have actually introduced a serious problem: What does it mean

$$\frac{\Gamma \vdash i_1 \doteq i_2}{\Gamma \vdash \tau(i_1) \leq \tau(i_2)}\ \text{iLR}\leq \qquad \frac{\Gamma \vdash i : \gamma \qquad \Gamma \vdash [i/a]A \leq B}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ \Pi\text{L}\leq \qquad \frac{\Gamma, b : \gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ \Pi\text{R}\leq$$

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

Figure 4: Subtyping and typing for indexed types (without term-level binders)

$$\frac{\Gamma \vdash i : \gamma \qquad \Gamma \vdash [i/b]e \Leftarrow A}{\Gamma \vdash \textbf{some}\ b : \gamma.\ e \Leftarrow A} \qquad \frac{\Gamma \vdash i : \gamma \qquad \Gamma \vdash [i/b]e \Rightarrow A}{\Gamma \vdash \textbf{some}\ b : \gamma.\ e \Rightarrow A}$$

Figure 5: Typing for the **some** binder

to mention an index variable a in an annotation when there are no term-level binders? The only thing that binds a is Π, and the scope of the binder $\Pi a{:}\gamma.\, A$ is just A. And what if the implicit condition in $\Pi\text{R}\leq$ and ΠI is triggered and we have to rename the variable? The user would be unable to refer to the variable in annotations.

One way to solve this is to introduce an odd sort of binding construct, **some** $a' : \gamma.\ e$, which binds its variable a' to some unwritten index expression—one chosen by the typechecker to make everything work out. An example:

$$\big(\lambda x. \ldots (\textbf{some}\ b : \gamma.\ x : \mathsf{list}(b * 2) >:> e) \ldots\big) \Leftarrow \Pi a{:}\mathsf{int}.\, \mathsf{list}(a * 2) \to \mathsf{list}(a)$$

Within the inner term e, we can write (right-hand) annotations that mention b: the typechecker chooses b to be a, which satisfies the guard condition $x \Leftarrow \mathsf{list}(b * 2)$.

The typing rules in Figure 5 substitute an index i for b in e, where i is well-sorted in the actual context Γ. Thus, all annotations that mention b will be renamed so they make sense under Γ. These rules do not require i to be a variable: the following code is acceptable, choosing i to be $a * 2$.

$$\big(\lambda x. \ldots (\textbf{some}\ b : \gamma.\ x : \mathsf{list}(b) >:> e) \ldots\big) \Leftarrow \Pi a{:}\mathsf{int}.\, \mathsf{list}(a * 2) \to \mathsf{list}(a)$$

Non-renaming substitutions achieve a measure of robustness: the type being checked against can, in some circumstances, change without requiring changes to internal annotations.

5.3 Indexed Types With Binders

Alternatively, we can have an explicit term-level introduction form for $\Pi a{:}\gamma.\, A$:

$$\frac{\Gamma, b : \gamma \vdash e \Leftarrow A}{\Gamma \vdash \Lambda b : \gamma.\ e \Leftarrow A}\ \Pi\text{I-explicit}$$

Dunfield and Pfenning (2004) did not take this route, because typing would fail for intersections of differently-quantified types. For example, the first conjunct of $(\Pi a{:}\gamma.\, A \to A') \wedge (B \to B')$ can type a term if it has a binder (for a), but the second conjunct cannot type a term with a binder (since $B \to B'$ has no Π). With our merge construct, we can write the term twice, with and without a binder.

$$\frac{}{(\cdot \vdash A) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-empty} \qquad \frac{\Gamma \vdash i : \gamma_0 \qquad ([i/a]\,\Gamma_0 \vdash [i/a]\,A_0) \lesssim (\Gamma \vdash A)}{(a{:}\gamma_0, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-ivar}$$

$$\frac{\Gamma \vdash \Gamma(x) \leq B_0 \qquad (\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}{(x{:}B_0, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-pvar}$$

$$\frac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : \ldots, (\Gamma_0 \vdash A_0), \ldots) \Rightarrow A}\ \text{ctx-anno}$$

Figure 6: Rules for contextual typing annotations

$$\begin{array}{rcl}
\mathsf{trans}(x) & = & x \\
\mathsf{trans}(())) & = & () \\
\mathsf{trans}(\lambda x.\, e) & = & \lambda x.\, \mathsf{trans}(e) \\
\mathsf{trans}(e_1\, e_2) & = & \mathsf{trans}(e_1)\ \mathsf{trans}(e_2) \\
\mathsf{trans}(e : (\Gamma_1 \vdash A_1), \ldots, (\Gamma_n \vdash A_n)) & = & \mathsf{trans}(\Gamma_1 \vdash A_1) \,„\, \ldots \,„\, \mathsf{trans}(\Gamma_n \vdash A_n) \\
 & & \text{where } \mathsf{trans}(d_1, \ldots, d_n \vdash A) = d_1 \mathbin{>\!:\!>} \ldots d_n \mathbin{>\!:\!>} (\mathsf{trans}(e) : A)
\end{array}$$

Figure 7: Translating contextual typing annotations

5.4 Free Annotation Revisited

Whether we have **some** binders or Λ binders, we maintain the property mentioned in Section 3.4: the user can always add an extra annotation if desired.

- If we have **some**, the user will need to add a **some** binder for any index variable mentioned in annotations (left- and right-hand).
- If we have Λ and rule ΠI-explicit instead of ΠI, the user must already have put in the Λ forms, and can refer to those bound index variables in annotations.

6 Comparison to Contextual Typing Annotations

We briefly review contextual typing annotations, introduced by Dunfield and Pfenning (2004). Such an annotation has a list As of typings $(\Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n)$. The typing rule ctx-anno (Figure 6) chooses a typing $\Gamma_0 \vdash A_0$ and then uses a *contextual subtyping relation* $(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$, which is derivable when Γ is at least as strong as Γ_0, that is, when Γ satisfies all assumptions listed in Γ_0. Declarations in Γ_0 thus should correspond to a sequence of guard annotations. Declarations of index variables in Γ_0, however, are treated differently: the rule $\lesssim$-ivar behaves like the typing rules for the **some** binder (Figure 5), effectively binding variables declared in Γ_0 so they can be used in A_0.

In hindsight, contextual typing annotations combine all the mechanisms in this paper—guard annotations, standard annotations, and merges: program variable declarations $x : A$ in Γ_0 correspond to a sequence of guard annotations, the type A_0 corresponds to a standard annotation, and the multiplicity of typings corresponds to merges. Translating contextual typing annotations (Figure 7) preserves typing:

Theorem 1 (Encoding Contextual Typing Annotations)**.**
If $\Gamma \vdash e \Leftarrow A$ *(resp.* $\Rightarrow$*) with rule* ctx-anno *available then* $\Gamma \vdash \mathsf{trans}(e') \Leftarrow A$ *(resp.* $\Rightarrow$*) without applying rule* ctx-anno*.*

Proof. By induction on the derivation. All cases are straightforward except when ctx-anno concludes the derivation.

In that case, apply the i.h. resulting in $\mathsf{trans}(e_0')$. This application of ctx-anno uses one of the contextual typings, say $(\Gamma_k \vdash A_k)$ where $\Gamma_k = d_1, \ldots, d_n$; the kth branch of the merge created by $\mathsf{trans}(-)$ is $d_1 >:> \ldots d_m >:> (\mathsf{trans}(e_0') : A)$.

By rule right-anno, $\Gamma \vdash \mathsf{trans}(e_0')A \Rightarrow A$.

By m applications of left-anno,

$$\Gamma \vdash\ d_1 >:> \ldots d_m >:> (\mathsf{trans}(e_0') : A)\ \ \Rightarrow\ A$$

Finally, apply merge$\Rightarrow$ as needed to pick out the kth branch of the merge created by $\mathsf{trans}(-)$. □

Given that we subsume contextual typing annotations, which approach should be preferred when designing a language? It is hard to give a universal answer. Generally speaking, simpler constructs are better than complicated ones, but fewer constructs are better than many. By the former criterion, the mechanisms proposed in this paper win; by the latter, contextual typing annotations win. The particular design setting matters: if we need some of these mechanisms already, their marginal cost is reduced. This was the case in the work that directly inspired this paper, elaboration-based typing of intersections and unions (Dunfield 2012), where the merge construct was already present.

7 Comparison to Contextual Types

There are several approaches to typing open code. In one such approach, contextual modal type theory (Nanevski et al. 2008), the contextual type $A\,[\Psi]$ represents data of type A closed under a context Ψ. Providing a substitution for the variables in Ψ allows a term of type $A\,[\Psi]$ to yield a term of type $A\,[\cdot]$, closed under the empty context—that is, a closed term.

Contextual types appear to subsume both guard annotations and our use of merges. For example, instead of the guard annotations in $\lambda x.\ \big(x : \mathsf{odd} >:> (x \cdot 1 : \mathsf{even})\big) \,„\, \big(x : \mathsf{even} >:> (x \cdot 1 : \mathsf{odd})\big)$ we could write

$$\lambda x.\ \textbf{let}\ r = (y \cdot 1) : \mathsf{even}\,[y : \mathsf{odd}] \wedge \mathsf{odd}\,[y : \mathsf{even}]\ \textbf{in}\ r\,[x/y]$$

Checking $(y \cdot 1)$ against the first conjunct of the (ordinary right-hand) annotation, $\mathsf{even}\,[y : \mathsf{odd}]$, shows that $(y \cdot 1)$ has type even when y is substituted with a value of type odd. The second conjunct is symmetric. In the body of the **let**, we plug in x. When we check the whole function against $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$, the variable x will have type odd in one subderivation of $\wedge$I, and type even in the other. In each subderivation, using intersection elimination gives r a contextual type that can be eliminated by substituting x for y.

Contextual types are versatile. For example, they enable us to lift the binding of r outside the function, and instantiate r with different concrete contexts (different substitutions for $y : \mathsf{even}$) at several program points. Extending typecheckers and compilers with such types, however, is nontrivial (Pientka 2008). Introducing contextual types just to support type annotations seems extravagant. If contextual types are already available in a language, of course, it could make sense to encode the annotation mechanisms of this paper as contextual types, or for programmers to write contextual types directly.

8 Implementation

Several of the ideas described above have been implemented in Stardust, a typechecker (and compiler) for a small language in the Standard ML tradition. In addition to intersection types and indexed types, Stardust supports union types, datasort refinements and parametric polymorphism.

The implementation, with some examples, can be downloaded from `http://stardust.qc.com`. The syntax diverges slightly from the above presentation:

- the left-hand annotation $\mathrm{d} :> e$ is written **where** d **do** e;
- type annotations can be given separately from their bindings; these annotations are similar to contextual type annotations, but with the $\mathrm{d} :> e$ syntax for variable typings;
- the **some** binder is (presently) only implemented for separate type annotations on bindings, not as an ordinary expression form.

An early version of Stardust was described in Dunfield (2007a,b), but the current version adds a number of important features, incorporating ideas from Dunfield (2009, 2012).

9 Usability

We briefly consider some practical issues around the usability of our annotation mechanisms.

The approach to bidirectional typechecking developed in Dunfield and Pfenning (2004) guarantees that right-hand annotations are needed only at redexes (most commonly, recursive function declarations). Once the user decides to add an annotation (whether strictly required for typechecking, or for the purpose of documentation), the next step—of adding a merge with left-hand annotations (or perhaps a contextual typing annotation)—is fully determined: if the term needs to have different types under different contexts, the user must add a merge and left-hand annotations.

The overall size of the annotations is hard to characterize. Some examples of annotated programs can be found in Xi (1998) for bidirectional typechecking with indexed types (but without intersections or contextual typings), Davies (2005) for bidirectional typechecking of refinement types and intersection types, and Dunfield (2007b) for bidirectional typechecking of refinement types, indexed types, intersection and union types. Our experience with our implementation is that for nontrivial uses of intersection and union types, the performance of typechecking becomes highly problematic long before the annotations become unacceptably long. It is difficult to see how truly complex annotations could be substantially reduced: if the annotations are complex, it is probably because the program specification is nontrivial.

Acknowledgments

I'd like to thank the ITRS reviewers for their comments and suggestions, for both the pre- and post-proceedings versions; the workshop participants for the discussion during the talk; Neelakantan R. Krishnaswami, for discussions and encouragement; Frank Pfenning, for so many things.

References

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. doi:10.1002/malq.19810270205.

Thierry Coquand, Gérard Huet, et al. Coq homepage, 2012. http://coq.inria.fr/.

Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110.

Joshua Dunfield. Refined typechecking with Stardust. In *Programming Languages meets Program Verification (PLPV '07)*, 2007a. doi:10.1145/1292597.1292602.

Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007b. CMU-CS-07-129.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML Workshop*, pages 15–26, 2009. doi:10.1145/1596627.1596631.

Joshua Dunfield. Elaborating intersection and union types. In *Int'l Conf. Functional Programming*, 2012. arXiv:1206.5386 [cs.PL].

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FoSSaCS '03)*, pages 250–266, 2003. doi:10.1007/3-540-36576-1_16.

Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *Principles of Programming Languages*, pages 281–292, 2004. doi:10.1145/964001.964025.

Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *Collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Computational Logic*, 9(3), 2008. doi:10.1145/1352582.1352591.

Frank Pfenning, Carsten Schürmann, et al. Twelf homepage, 2012. http://twelf.org/wiki/Main_Page.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Principles of Programming Languages*, pages 371–382, 2008. doi:10.1145/1328438.1328483.

Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-205.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Programming Languages and Systems*, 22:1–44, 2000. doi:10.1145/345099.345100.

John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, 1988. http://doi.library.cmu.edu/10.1184/OCLC/18612825.

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

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Principles of Programming Languages*, pages 214–227, 1999. doi:10.1145/292540.292560.