Untangling Typechecking of Intersections and Unions

Joshua Dunfield
School of Computer Science, McGill University
Montréal, Canada
joshua@cs.mcgill.ca

Intersection and union types denote conjunctions and disjunctions of properties. Using bidirectional typechecking, intersection types are relatively straightforward, but union types present challenges. For union types, we can case-analyze a subterm of union type when it appears in evaluation position (replacing the subterm with a variable, and checking that term twice under appropriate assumptions). This technique preserves soundness in a call-by-value semantics. Sadly, there are so many choices of subterms that a direct implementation is not practical. But carefully transforming programs into let-normal form drastically reduces the number of choices. The key results are soundness and completeness: a typing derivation (in the system with too many subterm choices) exists for a program if and only if a derivation exists for the let-normalized program.

1 Introduction

To check programs in advanced type systems, it can be useful to split the traditional typing judgment $e : A$ into two forms, $e \Uparrow A$ read "e synthesizes type A" and $e \Downarrow A$ read "e checks against type A", and requiring that the user write annotations on redexes. This *bidirectional typechecking* (Pierce and Turner 1998) is decidable for many interesting features, including intersection and union types without syntactic markers. *Tridirectional* typechecking (Dunfield and Pfenning 2004; Dunfield 2007b) is essentially bidirectional, but union types are eliminated with the aid of a *tridirectional rule* that uses an evaluation context $\mathscr{E}$:

$$\frac{\Gamma;\Delta_1 \vdash e' \Uparrow A \qquad \Gamma;\Delta_2,\overline{\mathbf{x}}{:}A \vdash \mathscr{E}[\overline{\mathbf{x}}] \Downarrow C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathscr{E}[e'] \Downarrow C}\ \mathsf{direct}\mathbb{L}$$

In this rule, Γ is an ordinary variable context and Δ_1,Δ_2 is the concatenation of *linear contexts*; linear variables $\overline{\mathbf{x}}$ in Δs essentially stand for subterms (occurrences) in the subject. $\mathsf{direct}\mathbb{L}$ gives e' a (linear) name $\overline{\mathbf{x}}$, so that a left rule, which decomposes types in the context Δ, can eliminate union types appearing in A. Instead of a direct union elimination rule like $\vee$E, we use $\mathsf{direct}\mathbb{L}$ together with a left rule $\vee\mathbb{L}$.

$$\frac{\Gamma \vdash e' \Uparrow A \vee B \qquad \Gamma,x{:}A \vdash \mathscr{E}[x] \Downarrow C \qquad \Gamma,y{:}B \vdash \mathscr{E}[y] \Downarrow C}{\Gamma \vdash \mathscr{E}[e'] \Downarrow C}\ [\vee\mathrm{E}] \qquad \frac{\Gamma;\Delta,\overline{\mathbf{x}}{:}A \vdash e \Downarrow C \qquad \Gamma;\Delta,\overline{\mathbf{x}}{:}B \vdash e \Downarrow C}{\Gamma;\Delta,\overline{\mathbf{x}}{:}A \vee B \vdash e \Downarrow C}\ \vee\mathbb{L}$$

While evaluation contexts are defined syntactically, this rule is not syntax-directed in the usual sense: many terms have more than one decomposition into some $\mathscr{E}[e']$ where the subterm e' can synthesize a type. Under a left-to-right (functions first, arguments second) call-by-value semantics, even $f\ x$ has three decompositions $\mathscr{E} = [\,]$, $\mathscr{E} = [\,]\ x$, $\mathscr{E} = f\ [\,]$, so a straightforward implementation of a system with $\mathsf{direct}\mathbb{L}$ would require far too much backtracking. Compounded with backtracking due to intersection and union types (e.g., if $f : (A_1 \to A_2) \wedge (B_1 \to B_2)$ we may have to try both $f : A_1 \to A_2$ and $f : B_1 \to B_2$), such an implementation would be hopelessly impractical.

This paper reformulates tridirectional typechecking (summarized in Section 2) to work on terms in a particular *let-normal form*, in which steps of computation are sequenced and intermediate computations

Submitted to:
2010 Workshop on Intersection Types and Related Systems (ITRS '10)

are named. The let-normal transformation (Section 3) drastically constrains the decomposition by sequencing terms, forcing typechecking to proceed left to right (with an interesting exception). The results stated in Section 4 guarantee that the let-normal version of a program e is well typed under the let-normal version of the type system if and only if e is well typed under the tridirectional system.

The let-normal transformation itself is not complicated, though the motivation for my particular formulation is somewhat involved. The details of the transformation may be of interest to designers of advanced type systems, whether their need for a sequential form arises from typechecking itself (as in this case) or from issues related to compilation.

Unfortunately, the proofs (especially the proof of completeness) are very involved; I couldn't even fit all the statements of lemmas in this paper, much less sketch their proofs. I hope only to convey a shadow of the argument's structure.

This paper distills part of my dissertation (Dunfield 2007b, chapter 5). To simplify presentation, I omit tuples, datasort refinements, indexed types (along with universal and existential quantification, guarded types, and asserting types), and a greatest type $\top$.

2 Tridirectional Typechecking

We have functions, products, intersections, unions, and an empty type $\bot$. We'll use a unit type and other base types like int in examples. In the terms e, we have variables x (which are values) bound by $\lambda x.e$, variables u (not values) bound by $\mathsf{fix}\, u.e$, and call-by-value application $e_1\, e_2$. Note the lack of syntactic markers for intersections or unions. As usual, $\mathscr{E}[e']$ is the evaluation context $\mathscr{E}$ with its hole replaced by e'. To replace x with e_1, we write $[e_1/x]e_2$: "e_1 for x in e_2".

$$
\begin{array}{rrcl}
\text{Types} & A,B,C,D & ::= & A \to B \mid A \wedge B \mid A \vee B \mid \bot \\
\text{Terms} & e & ::= & x \mid u \mid \lambda x.e \mid e_1\, e_2 \mid \mathsf{fix}\, u.e \\
\text{Values} & v & ::= & x \mid \lambda x.e \\
\text{Evaluation contexts} & \mathscr{E} & ::= & [] \mid \mathscr{E}\, e \mid v\, \mathscr{E}
\end{array}
$$

Small-step reduction rules

$$
\begin{array}{rcl}
\mathscr{E}[(\lambda x.e)\, v] & \mapsto & \mathscr{E}[[v/x]e] \\
\mathscr{E}[\mathsf{fix}\, u.e] & \mapsto & \mathscr{E}[[(\mathsf{fix}\, u.e)\,/\,u]e]
\end{array}
$$

We'll start by looking at the "left tridirectional" (in this paper, called just "tridirectional") system. This system was presented in Dunfield and Pfenning (2004) and Dunfield (2007b, chapter 4); space allows only a cursory description.

The subtyping judgment (Figure 1) is $A \leq B$. Transitivity is admissible. $\wedge$ does not distribute across $\to$, for reasons explained by Davies and Pfenning (2000).

Figure 2 gives the typing rules. The judgment $\Gamma;\Delta \vdash e \Uparrow A$ is read "e synthesizes type A", and $\Gamma;\Delta \vdash e \Downarrow A$ is read "e checks against A". When synthesizing, A is output; when checking, A is input.

$$
\begin{array}{rrcl}
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Linear contexts} & \Delta & ::= & \cdot \mid \Delta, \overline{\mathbf{x}}{:}A
\end{array}
$$

Contexts Γ have regular variable declarations. Linear contexts $\Delta ::= \cdot \mid \Delta, \overline{\mathbf{x}}{:}A$ have linear variables. If $\Gamma;\Delta \vdash e\ldots$ is derivable, then "$\Delta \Vdash e\ \mathsf{ok}$", read "$e$ OK under Δ": each $\overline{\mathbf{x}}$ declared in Δ appears exactly once in e, and e contains no other linear variables. Rules that decompose the subject, such as $\to$E decomposing $e_1\, e_2$ into e_1 and e_2, likewise decompose Δ.

Most rules follow a formula devised in Dunfield and Pfenning (2004): introduction rules, such as $\to$I, check; elimination rules, such as $\to$E, synthesize. Introduction forms like $\lambda x.e$ thus construct *synthesizing terms*, while elimination forms like $e_1\, e_2$ are *checked terms*. Some rules fall outside this classification. The *assumption* rules var, $\overline{\mathsf{var}}$ and fixvar synthesize (an assumption $x{:}A$ can be read $x \Uparrow A$). The *subsumption* rule sub allows a term that synthesizes A to check against a type B, provided A is

a subtype of B. The rule ctx-anno permits *contextual typing annotations*; for example, in $(\mathsf{succ}\ x : (x{:}\mathsf{odd} \vdash \mathsf{even}, x{:}\mathsf{even} \vdash \mathsf{odd}))$, the annotated term $\mathsf{succ}\ x$ is checked against even if $x{:}\mathsf{odd} \in \Gamma$, and against odd if $x{:}\mathsf{even} \in \Gamma$. The premise $(\Gamma_0 \vdash A) \lesssim (\Gamma \vdash A)$ is derivable if the assumptions in Γ support the assumptions in Γ_0. For details, see Dunfield and Pfenning (2004).

Finally, we have *left rules* $\wedge\mathbb{L}_1$, $\wedge\mathbb{L}_2$, $\vee\mathbb{L}$, $\bot\mathbb{L}$ which act on linear assumptions $\overline{\mathbf{x}}{:}A$ where A is of intersection, union, or empty type. These act as elimination rules—for $\vee$ and $\bot$, they are the *only* elimination rules. $\wedge\mathbb{L}_1$ and $\wedge\mathbb{L}_2$ are not useful alone (the ordinary eliminations $\wedge\mathrm{E}_1$ and $\wedge\mathrm{E}_2$ would do) but are needed to expose a nested $\vee$ for $\vee\mathbb{L}$, or a $\bot$ for $\bot\mathbb{L}$.

The backtracking required to choose between $\wedge\mathrm{E}_1$ and $\wedge\mathrm{E}_2$, or between $\vee\mathrm{I}_1$ and $\vee\mathrm{I}_2$, or between the related subtyping rules, as well as the need to check a single term more than once ($\wedge$I, $\vee\mathbb{L}$) suggests that typechecking is exponential. In fact, Reynolds (1996, pp. 67–68) proved that for a closely related system, typechecking is PSPACE-hard. We can't make typechecking polynomial, but "untangling" directL will remove one additional source of complexity.

$$\boxed{A \leq B} \quad \frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2} \to\leq \quad \frac{A_1 \leq B}{A_1 \wedge A_2 \leq B} \wedge\mathrm{L}_1\leq \quad \frac{A_2 \leq B}{A_1 \wedge A_2 \leq B} \wedge\mathrm{L}_2\leq \quad \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \wedge B_2} \wedge\mathrm{R}\leq$$

$$\frac{}{\bot \leq A} \bot\mathrm{L}\leq \quad \frac{A_1 \leq B \qquad A_2 \leq B}{A_1 \vee A_2 \leq B} \vee\mathrm{L}\leq \quad \frac{A \leq B_1}{A \leq B_1 \vee B_2} \vee\mathrm{R}_1\leq \quad \frac{A \leq B_2}{A \leq B_1 \vee B_2} \vee\mathrm{R}_2\leq$$

Figure 1: Subtyping

$$\frac{\Gamma(x) = A}{\Gamma;\cdot \vdash x \Uparrow A}\ \mathsf{var} \qquad \frac{}{\Gamma;\overline{\mathbf{x}}{:}A \vdash \overline{\mathbf{x}} \Uparrow A}\ \overline{\mathsf{var}} \qquad \frac{\Gamma, x{:}A;\cdot \vdash e \Downarrow B}{\Gamma;\cdot \vdash \lambda x.e \Downarrow A \to B} \to\mathrm{I} \qquad \frac{\Gamma;\Delta_1 \vdash e_1 \Uparrow A \to B \qquad \Gamma;\Delta_2 \vdash e_2 \Downarrow A}{\Gamma;\Delta_1,\Delta_2 \vdash e_1 e_2 \Uparrow B} \to\mathrm{E}$$

$$\frac{\Gamma;\Delta \vdash e \Uparrow A \qquad A \leq B}{\Gamma;\Delta \vdash e \Downarrow B}\ \mathsf{sub} \qquad \frac{\Gamma(u) = A}{\Gamma;\cdot \vdash u \Uparrow A}\ \mathsf{fixvar} \qquad \frac{\Gamma, u{:}A;\cdot \vdash e \Downarrow A}{\Gamma;\cdot \vdash \mathsf{fix}\, u.e \Downarrow A}\ \mathsf{fix}$$

$$\frac{\Gamma \vdash e\ \mathsf{ok} \qquad \Delta, \overline{\mathbf{x}}{:}\bot \Vdash e\ \mathsf{ok}}{\Gamma;\Delta, \overline{\mathbf{x}}{:}\bot \vdash e \Downarrow C} \bot\mathbb{L} \qquad \frac{(\Gamma_0 \vdash A) \lesssim (\Gamma \vdash A) \qquad \Gamma;\Delta \vdash e \Downarrow A}{\Gamma;\Delta \vdash (e : \ldots, (\Gamma_0 \vdash A), \ldots) \Uparrow A}\ \mathsf{ctx\text{-}anno}$$

$$\frac{\Gamma;\Delta, \overline{\mathbf{x}}{:}A \vdash e \Downarrow C}{\Gamma;\Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \Downarrow C} \wedge\mathbb{L}_1 \qquad \frac{\Gamma;\Delta \vdash v \Downarrow A \qquad \Gamma;\Delta \vdash v \Downarrow B}{\Gamma;\Delta \vdash v \Downarrow A \wedge B} \wedge\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash e \Uparrow A \wedge B}{\Gamma;\Delta \vdash e \Uparrow A} \wedge\mathrm{E}_1$$

$$\frac{\Gamma;\Delta, \overline{\mathbf{x}}{:}B \vdash e \Downarrow C}{\Gamma;\Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \Downarrow C} \wedge\mathbb{L}_2 \qquad \frac{\Gamma;\Delta \vdash e \Uparrow A \wedge B}{\Gamma;\Delta \vdash e \Uparrow B} \wedge\mathrm{E}_2$$

$$\frac{\Gamma;\Delta, \overline{\mathbf{x}}{:}A \vdash e \Downarrow C \qquad \Gamma;\Delta, \overline{\mathbf{x}}{:}B \vdash e \Downarrow C}{\Gamma;\Delta, \overline{\mathbf{x}}{:}A \vee B \vdash e \Downarrow C} \vee\mathbb{L} \qquad \frac{\Gamma;\Delta \vdash e \Downarrow A}{\Gamma;\Delta \vdash e \Downarrow A \vee B} \vee\mathrm{I}_1 \qquad \frac{\Gamma;\Delta \vdash e \Downarrow B}{\Gamma;\Delta \vdash e \Downarrow A \vee B} \vee\mathrm{I}_2$$

$$\frac{\Gamma;\Delta_1 \vdash e' \Uparrow A \qquad \Gamma;\Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathscr{E}[\overline{\mathbf{x}}] \Downarrow C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathscr{E}[e'] \Downarrow C}\ \mathsf{directL} \quad \text{where } e' \text{ is not a linear variable}$$

Figure 2: The left tridirectional system

2.1 Tridirectional typechecking and evaluation contexts

Rule directL's use of an evaluation context might give the impression that typechecking simply proceeds in the order in which terms are actually evaluated. However, this is not the case. The subject of directL is $\mathscr{E}[e']$ where e' synthesizes a type, so certainly e' must be in *an* evaluation position, but there may be

several such positions. Even a term as simple as $f\ (x\ y)$ has 5 subterms in evaluation position, each corresponding to a different evaluation context $\mathscr{E}$:

$$\mathscr{E} = []\ (x\ y) \quad \text{and} \quad e' = f \qquad \mathscr{E} = f\ (x\ []) \quad \text{and} \quad e' = y \qquad \mathscr{E} = f\ [] \quad \text{and} \quad e' = (x\ y)$$
$$\mathscr{E} = f\ ([]\ y) \quad \text{and} \quad e' = x \qquad\qquad\qquad\qquad\qquad\qquad \mathscr{E} = [] \quad \text{and} \quad e' = f\ (x\ y)$$

In fact, we may need to repeatedly apply direct$\mathbb{L}$ to the same subject term with different choices of $\mathscr{E}$! For example, we might use $\mathscr{E} = []\ (x\ y)$ to name an f of union type, introducing $\overline{\mathbf{f}}{:}A \vee B$ into the context; then, case-analyze $A \vee B$ with $\vee$E; finally, choose $\mathscr{E} = \overline{\mathbf{f}}\ ([]\ y)$ to name x (also of union type). Thus we are faced not with a choice over decompositions, but over many *sequences* of decompositions.

Typechecking cannot go strictly left to right. Given an ML-like `int option` type, containing `None` and some integer `Some`(n), assume `None` : none and `Some`(n) : some. Then, if map f (`Some`(n)) returns `Some`$(f\ n)$ and map f `None` is none, then map : (int $\to$ int) $\to$ ((some $\to$ some) $\wedge$ (none $\to$ none)). Similarly, a function filtering out negative integers could have type filter : int $\to$ (some $\vee$ none).

Consider the term (map f) (filter n). The term (map f) synthesizes (some $\to$ some)$\wedge$(none $\to$ none). This is an intersection type—we'll abbreviate it as (s$\to$s)$\wedge$(n$\to$n)—and the intersection must be eliminated so that rule $\to$E can be applied to (map f) (filter x). However, we cannot commit to one part of the intersection yet, because we must first case-analyze the union type of the subterm (filter x). We need to "jump over" (map f) to type (filter x), so apply direct$\mathbb{L}$ with evaluation context $\mathscr{E} = []$ (filter x), giving (map f) the name $\overline{\mathbf{x}}$; second, apply direct$\mathbb{L}$ with context $\mathscr{E} = \overline{\mathbf{x}}\ []$, synthesizing some $\vee$ none for (filter x). Rule $\vee\mathbb{L}$ splits on $\overline{\mathbf{y}}$:some$\vee$none; in its left subderivation $\mathscr{D}_{\mathsf{some}}$, we have $\overline{\mathbf{y}}$:some, so $\wedge$E$_1$ on $\overline{\mathbf{x}} \Uparrow$ (some $\to$ some) $\wedge$ (none $\to$ none) gives $\overline{\mathbf{x}} \Uparrow$ some $\to$ some, while its right subderivation $\mathscr{D}_{\mathsf{none}}$ has $\overline{\mathbf{y}}$:none, so $\wedge$E$_2$ gives $\overline{\mathbf{x}} \Uparrow$ none $\to$ none. Writing Δ for $\overline{\mathbf{x}}$:((s$\to$s)$\wedge$(n$\to$n)), the derivation is

$$\dfrac{\vdash \mathit{map}\ f \Uparrow (\mathsf{s}\to\mathsf{s})\wedge(\mathsf{n}\to\mathsf{n}) \qquad \dfrac{\Delta \vdash \mathsf{filter}\ x \Uparrow \mathsf{some} \vee \mathsf{none} \qquad \dfrac{\mathscr{D}_{\mathsf{some}} \qquad \mathscr{D}_{\mathsf{none}}}{\Delta, \overline{\mathbf{y}}{:}(\mathsf{some}\vee\mathsf{none}) \vdash \overline{\mathbf{x}}\ \overline{\mathbf{y}} \Downarrow C}\ \vee\mathbb{L}}{\Delta \vdash \overline{\mathbf{x}}\ (\mathsf{filter}\ x) \Downarrow C}\ \mathsf{direct}\mathbb{L}}{\vdash (\mathit{map}\ f)\ (\mathsf{filter}\ x) \Downarrow C}\ \mathsf{direct}\mathbb{L}$$

where C is some $\vee$ none, and the derivations $\mathscr{D}_{\mathsf{some}}$ and $\mathscr{D}_{\mathsf{none}}$ are

$$\dfrac{\dfrac{\ldots \vdash \overline{\mathbf{x}} \Uparrow (\mathsf{some}\to\mathsf{some})\wedge(\mathsf{none}\to\mathsf{none})}{\ldots \vdash \overline{\mathbf{x}} \Uparrow \mathsf{some} \to \mathsf{some}}\ \wedge\mathrm{E}_1}{\vdots}{\Delta, \overline{\mathbf{y}}{:}\mathsf{some} \vdash \overline{\mathbf{x}}\ \overline{\mathbf{y}} \Downarrow C} \qquad \text{and} \qquad \dfrac{\dfrac{\ldots \vdash \overline{\mathbf{x}} \Uparrow (\mathsf{some}\to\mathsf{some})\wedge(\mathsf{none}\to\mathsf{none})}{\ldots \vdash \overline{\mathbf{x}} \Uparrow \mathsf{none} \to \mathsf{none}}\ \wedge\mathrm{E}_2}{\vdots}{\Delta, \overline{\mathbf{y}}{:}\mathsf{none} \vdash \overline{\mathbf{x}}\ \overline{\mathbf{y}} \Downarrow C}$$

On a purely theoretical level, the tridirectional system is acceptable, but the nondeterminism is excessive. Xi approached (very nearly) the same problem by transforming the program so the term of $\vee$ type appears before the term of $\wedge$ type. (Actually, Xi had index-level quantifiers Σ and Π instead of $\vee$ and $\wedge$, but these are analogous.) A standard let-normal translation $|e|$ (Xi 1998, p. 86), where $|e_1\ e_2| = \mathsf{let}\, x_1 = |e_1|\, \mathsf{in}\, \mathsf{let}\, x_2 = |e_2|\, \mathsf{in}\, x_1 x_2$ suffices for the examples above. (In Xi's system, existential variables are unpacked where a term of existential type is let-bound: an existential variable b' is unpacked at the binding of x_2, which appears before the application $x_1 x_2$ at which the universal variable a must be instantiated.) Unfortunately, the translation interacts unpleasantly with bidirectionality: terms such as map $(\lambda x.e)$, in which $(\lambda x.e)$ must be checked, no longer typecheck because the λ becomes the right hand side of a let, in $\mathsf{let}\, x_1 = \mathsf{map}\, \mathsf{in}\, \mathsf{let}\, x_2 = \lambda x.e\, \mathsf{in}\, x_1 x_2$ and let-bound expressions must synthesize a type, but $\lambda x.e$ does not. Typechecking becomes incomplete in the sense that some programs that were well typed are not well typed after translation.

Xi ameliorated this incompleteness by treating $e_1\, v_2$ as a special case (Xi 1998, p. 139): $|e_1\ v_2| = \mathsf{let}\, x_1 = |e_1|\, \mathsf{in}\, x_1\, v_2$. Now v_2 (which is $\lambda x.e$ in the above example) is in a checking position. This is

$$\frac{\Gamma;\Delta_1 \vdash e' \Uparrow A \qquad \Gamma;\Delta_2,\overline{\mathbf{x}}{:}A \vdash \mathscr{Q}[\overline{\mathbf{x}}] \Downarrow C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathsf{let}\,\overline{\mathbf{x}} = e' \,\mathsf{in}\, \mathscr{Q}[\overline{\mathbf{x}}] \Downarrow C}\,\mathsf{let} \qquad \frac{\Gamma;\Delta, {\sim}\overline{\mathbf{x}} = v \vdash \mathscr{Q}[\overline{\mathbf{x}}] \Downarrow C}{\Gamma;\Delta \vdash \mathsf{let} {\sim}\overline{\mathbf{x}} = v \,\mathsf{in}\, \mathscr{Q}[\overline{\mathbf{x}}] \Downarrow C}\,\mathsf{let}{\sim} \qquad \frac{\Gamma;\Delta_1 \vdash v \Uparrow A \qquad \Gamma;\Delta_2,\overline{\mathbf{x}}{:}A \vdash e \Downarrow C}{\Gamma;\Delta_1,\Delta_2, {\sim}\overline{\mathbf{x}} = v \vdash e \Downarrow C}\,{\sim}\mathsf{var}$$

. . . plus all rules in Figure 2, except direct$\mathbb{L}$

Figure 3: The *let-normal type system* for terms containing let $\overline{\mathbf{x}}$ bindings

adequate for non-synthesizing values, but terms such as map (case z of $\ldots$), where a non-synthesizing non-value is in checking position, remain untypable. It is not clear why Xi did not also have special cases for case and other non-synthesizing non-values, e.g. $|e_1\ (\mathsf{case}\ e\ \mathsf{of}\ ms)| = \mathsf{let}\, x_1 = |e_1|\,\mathsf{in}\, x_1\,|\mathsf{case}\ e\ \mathsf{of}\ ms|$. Xi's translation is also incomplete for terms like $f\ (\mathsf{case}\ x\ \mathsf{of}\ ms)$. Suppose x synthesizes a union that must be analyzed to select the appropriate part of an intersection in the type of f. Since x's scope—and thus the scope of its union—is entirely within the let created for the case, typechecking fails.

$$\begin{aligned} |f\ (\mathsf{case}\ x\ \mathsf{of}\ ms)| &= \mathsf{let}\, f_1 = f \,\mathsf{in}\, \mathsf{let}\, x_0 = |\mathsf{case}\ x\ \mathsf{of}\ ms| \,\mathsf{in}\, f_1\ x_0 \\ &= \mathsf{let}\, f_1 = f \,\mathsf{in}\, \mathsf{let}\, x_0 = \big(\mathsf{let}\, x_1 = x \,\mathsf{in}\, \mathsf{case}\ x_1\ \mathsf{of}\ |ms|\big) \,\mathsf{in}\, f_1\ x_0 \end{aligned}$$

It could be argued that the cases in which Xi's translation fails are rare in practice. However, that may only increase confusion when such a case is encountered. I follow Xi's general approach of sequentializing the program before typechecking, but no programs are lost in my translation.

Do we need all the freedom that direct$\mathbb{L}$ provides? No. At the very least, if we do not need to name a subterm, naming it anyway does no harm. But naming all the subterms only slightly reduces the nondeterminism. Clearly, a strategy of in-order traversal is sound (we can choose to apply direct$\mathbb{L}$ from left to right if we like). It is tempting to think it is complete. In fact, it holds for many programs, but fails for a certain class of annotated terms. We will explain why as we present the general mechanism for enforcing a strategy of left-to-right traversal except for certain annotated terms.

3 Let-Normal Typechecking

We'll briefly mention previous work on let-normal form, then explain the ideas behind the variant here, including why we need a *principal synthesis of values* property. Because the most universal form of principality does not hold for a few terms, we introduce *slack bindings*.

Traditional let-normal or A-normal transformations (Moggi 1988; Flanagan et al. 1993) (1) explicitly sequence the computation, and (2) name the result of each intermediate computation. (Continuation-passing style (CPS) (Reynolds 1993) also (3) introduces named continuations. Thus let-normal form is also known as *two-thirds CPS*.) Many compilers for functional languages use some kind of let-normal form to facilitate optimizations; see, for instance, Tarditi et al. (1996), Reppy (2001), Chlipala et al. (2005), and Peyton Jones et al. (2006).

Our let-normal form will sequentialize the computation, but it does not only name intermediate *computations*, but *values* as well. In our let-normal type system, direct$\mathbb{L}$ is replaced by a rule let that can *only* be applied to let; see Figure 3. $\mathscr{Q}$ is a special evaluation context, discussed below.

This let is a syntactic marker with no computational character. In contrast to let-normal translations for compilation purposes, there is no evaluation step (reduction) corresponding to a let. I won't even give a dynamic semantics for terms with lets. It would be easy; it's simply not useful here. If we insist on knowing what a let-normal term e means, we can use a standard call-by-value operational semantics over the term's reverse translation.

Instead of making explicit the order of computation, our let-normal form makes explicit the order of *typechecking*—or rather, the order in which direct$\mathbb{L}$ names subterms in evaluation position. Thus, to be

complete with respect to the tridirectional system, the transformation must create a let for every subterm in synthesizing form: if an (untranslated) program contains a subterm e' in synthesizing form, it might be possible to name e' with direct$\mathbb{L}$, so the let-normal translation must bind e'. Otherwise, a chance to apply $\vee\mathbb{L}$ is lost. Even variables x must be named, since they synthesize a type and so can be named in direct$\mathbb{L}$. This models an "aggressive" strategy of applying direct$\mathbb{L}$. On the other hand, checked terms like $\lambda x.e$ can't synthesize, so we won't name them.

Another consequence of the let-normal form following typing, not evaluation, is that $\mathsf{let}\,\overline{\mathbf{x}}=v_1\,\mathsf{in}\,v_2$ is considered a value—after all, the original term $[v_1/\overline{\mathbf{x}}]v_2$ was a value, and we transformed a value into a non-value we could not apply value-restricted typing rules such as $\wedge$I, leading to incompleteness.

We define the translation by a judgment $e \hookrightarrow L+e'$, read "e translates to a sequence of let-bindings L with body e'". For example, the translation of $f\ (x\ y)$, which names every synthesizing subterm, is

$$\mathsf{let}\,\overline{\mathbf{f}}=f\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{x}}=x\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{y}}=y\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{z}}=\overline{\mathbf{x}}\ \overline{\mathbf{y}}\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{a}}=\overline{\mathbf{f}}\ \overline{\mathbf{z}}\,\mathsf{in}\,\overline{\mathbf{a}}$$

This is expressed by the judgment $f\ (x\ y) \hookrightarrow \overline{\mathbf{f}}=f,\overline{\mathbf{x}}=x,\overline{\mathbf{y}}=y,\overline{\mathbf{z}}=\overline{\mathbf{x}}\ \overline{\mathbf{y}},\overline{\mathbf{a}}=\overline{\mathbf{f}}\ \overline{\mathbf{z}}+\overline{\mathbf{a}}$. Figure 4 has the definition. Note that $L+e'$ is not a term; $+$ is punctuation. We write $L\ \mathsf{in}\ e'$ as shorthand: read $e \hookrightarrow L+e'$ as "$e \hookrightarrow L\ \mathsf{in}\ e'$". The divergent notations come from the multiple decompositions of a term into a pair of bindings and a "body". For example, $\mathsf{let}\,\overline{\mathbf{x}}_1=e_1\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{x}}_2=e_2\,\mathsf{in}\,e_3$ can be written three ways: **(1)** $\cdot\ \mathsf{in}\ \mathsf{let}\,\overline{\mathbf{x}}_1=e_1\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{x}}_2=e_2\,\mathsf{in}\,e_3$, **(2)** $(\overline{\mathbf{x}}_1=e_1)\ \mathsf{in}\ \mathsf{let}\,\overline{\mathbf{x}}_2=e_2\,\mathsf{in}\,e_3$, or **(3)** $(\overline{\mathbf{x}}_1=e_1,\overline{\mathbf{x}}_2=e_2)\ \mathsf{in}\ e_3$. The last decomposition is *maximal*: it has the maximum number of bindings (and the smallest 'body'), which is the case when the body isn't a let. If $e \hookrightarrow L+e'$ then $L\ \mathsf{in}\ e'$ is maximal.

Again, to model a complete strategy of direct$\mathbb{L}$-application, in $e \hookrightarrow L+e'$ we need L to bind all the synthesizing subterms that could be in evaluation position (after applying direct$\mathbb{L}$ zero or more times).

We syntactically partition terms into *pre-* and *anti*-values. A *pre-value* $\check{e}$ is a value, such as x, or a term that can "become" a value via direct$\mathbb{L}$, such as $x\ y$ which "becomes" the value $\overline{\mathbf{z}}$ in the derivation. (The háček $\check{}$ above the e is shaped like a 'v' for 'value'.) An *anti-value* $\widehat{e}$, such as $\mathsf{fix}\,u.e$ (or $\mathsf{case}\ e\ \mathsf{of}\ ms$) is not a value and cannot become a value.

direct$\mathbb{L}$ can replace any synthesizing subterm with a linear variable, so the pre-values must include both the values and the synthesizing forms. This leads to the following grammar for pre-values, with values x, $\overline{\mathbf{x}}$, and $\lambda x.e$ and synthesizing forms $(e : As)$, e_1e_2, u. (In the full system, the prevalues also include checking forms that can become values if all their subterms can, such as (e_1,e_2).)

$$\begin{array}{rrcl} \text{Pre-values} & \check{e} & ::= & x \mid \overline{\mathbf{x}} \mid (e : As) \mid \lambda x.e \mid e_1e_2 \mid u \\ \text{Anti-values} & \widehat{e} & ::= & \mathsf{fix}\,u.e \end{array}$$

The distinction matters for terms with sequences of immediate subterms such that at least two subterms in the sequence may be in evaluation position. Only application e_1e_2 has this property (and in the full system, pairs (e_1,e_2)). $\lambda x.e$ and $\mathsf{fix}\,u.e$ have no subterms in evaluation position at all.

A telling example is $(\mathsf{fix}\,u.e)\ (\omega\ x)$ where $\omega : \ldots \to \bot$. In the tridirectional system, this term has no synthesizing subterms in evaluation position. In particular, $\omega\ x$ is not in evaluation position, so however we translate the term, we must not bind $\omega\ x$ outside the outer application; if we did, we would add $\overline{\mathbf{z}}{:}\bot$ to the context and could apply rule $\bot\mathbb{L}$ to declare the outer application well typed while ignoring e! If e is ill-typed, this is actually unsound. On the other hand, in the term $(f\ g)\ (\omega\ x)$ the left tridirectional system *can* bind $\omega\ x$ before checking the pair, by applying direct$\mathbb{L}$ with $\mathscr{E} = []\ (\omega\ x)$ (synthesizing a type for $f\ g$, ensuring soundness) to yield a subject $\overline{\mathbf{x}}\ (\omega\ x)$ in which $\omega\ x$ is in evaluation position.

The difference is that $\mathsf{fix}\,u.e$ is an anti-value, while $f\ g$ is a pre-value. Therefore, given an application $e_1\ e_2$, if e_1 is some anti-value $\widehat{e}_1$, the translation places the bindings for subterms of e_2 (e.g. $\overline{\mathbf{z}}=\omega\ x$ above) *inside* the argument part. On the other hand, if e_1 is a pre-value $\check{e}_1$, the translation puts the bindings for subterms of e_2 *outside* the application. See the shaded rules in Figure 4.

$\boxed{e \hookrightarrow L + e'}$ read "e translates to bindings L with result e'"

$$\frac{}{x \hookrightarrow (\overline{\mathbf{x}} = x) + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e'}{\lambda x.e \hookrightarrow \cdot + \lambda x.(L \text{ in } e')} \qquad \frac{}{u \hookrightarrow (\overline{\mathbf{x}} = u) + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e'}{\mathsf{fix}\, u.e \hookrightarrow \cdot + \mathsf{fix}\, u.(L \text{ in } e')}$$

$$\frac{\widehat{e_1} \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{\widehat{e_1} e_2 \hookrightarrow L_1, \overline{\mathbf{x}} = e_1'(L_2 \text{ in } e_2') + \overline{\mathbf{x}}} \qquad \frac{\check{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{\check{e}_1 e_2 \hookrightarrow L_1, L_2, \overline{\mathbf{x}} = e_1' e_2' + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e' \qquad e \text{ not a value}}{(e : As) \hookrightarrow L, \overline{\mathbf{x}} = (e' : As) + \overline{\mathbf{x}}}$$

$$\frac{}{\overline{\mathbf{x}} \hookrightarrow \cdot + \overline{\mathbf{x}}} \qquad \frac{v \hookrightarrow L + e'}{(v : As) \hookrightarrow L, \sim\overline{\mathbf{x}} = (e' : As) + \overline{\mathbf{x}}}$$

Figure 4: The let-normal transformation

Elongated evaluation contexts $\mathscr{Q}$, unlike ordinary evaluation contexts $\mathscr{E}$, can skip over pre-values. $\mathscr{Q}$ is a sort of transitive closure of $\mathscr{E}$: if, by repeatedly replacing pre-values in evaluation position with values, some subterm is then in evaluation position, that subterm is in elongated evaluation position. In a sequence of direct$\mathbb{L}$-applications, subterms in evaluation position are replaced with linear variables, which are values. For example, z is not in evaluation position in $(x\ y)\ z$, but applying direct$\mathbb{L}$ with $\mathscr{E} = []\ z$ yields a subderivation with subject $\overline{\mathbf{x}}\ z$, in which z *is* in evaluation position. A $\mathscr{Q}$ is thus a path that can skip pre-values: if every intervening subterm is a pre-value (equivalently, if there is no intervening anti-value), the hole is in elongated evaluation position. The grammar for let-normal terms ensures that the body e_2 of $\mathsf{let}\,\overline{\mathbf{x}} = e_1\,\mathsf{in}\,e_2$ must have the form $\mathscr{Q}[\overline{\mathbf{x}}]$.

$$\begin{array}{rrcl}
\text{Elongated} & \mathscr{Q} & ::= & [] \mid \mathscr{Q}e \mid \check{e}\mathscr{Q} \mid (\mathscr{Q} : As) \\
\text{evaluation contexts} & & & \mid \mathsf{let}\,\overline{\mathbf{x}} = \mathscr{Q}\,\mathsf{in}\,e \mid \mathsf{let}\,\overline{\mathbf{x}} = \check{e}\,\mathsf{in}\,\mathscr{Q} \mid \mathsf{let} \sim\overline{\mathbf{x}} = \mathscr{Q}\,\mathsf{in}\,e \mid \mathsf{let} \sim\overline{\mathbf{x}} = v\,\mathsf{in}\,\mathscr{Q} \\
\text{Terms} & e & ::= & \ldots \mid \mathsf{let}\,\overline{\mathbf{x}} = e_1\,\mathsf{in}\,\mathscr{Q}[\overline{\mathbf{x}}] \mid \mathsf{let} \sim\overline{\mathbf{x}} = v_1\,\mathsf{in}\,\mathscr{Q}[\overline{\mathbf{x}}] \\
\text{Values} & v & ::= & x \mid \lambda x.e \mid \overline{\mathbf{x}} \mid \mathsf{let}\,\overline{\mathbf{x}} = v_1\,\mathsf{in}\,v_2 \mid \mathsf{let} \sim\overline{\mathbf{x}} = v_1\,\mathsf{in}\,v_2 \\
\text{Eval. contexts} & \mathscr{E} & ::= & \ldots \mid \mathsf{let}\,\overline{\mathbf{x}} = \mathscr{E}\,\mathsf{in}\,e \mid \mathsf{let}\,\overline{\mathbf{x}} = v\,\mathsf{in}\,\mathscr{E} \mid \mathsf{let} \sim\overline{\mathbf{x}} = \mathscr{E}\,\mathsf{in}\,e \mid \mathsf{let} \sim\overline{\mathbf{x}} = v\,\mathsf{in}\,\mathscr{E} \\
\text{Sequences of bindings} & L & ::= & \cdot \mid L, (\overline{\mathbf{x}} = e) \mid L, (\sim\overline{\mathbf{x}} = v)
\end{array}$$

3.1 Principal synthesis of values

A key step in completeness is the movement of let-bindings outward. To prove this preserves typing, we show that principal types (Hindley 1969) exist in certain cases. Consider the judgment $x : (A_1 \to B) \wedge (A_2 \to B), y : A_1 \vee A_2; \cdot \vdash x\ y \Downarrow B$. To derive this in the left tridirectional system, we need direct$\mathbb{L}$ with $\mathscr{E} = x\ []$ to name y as a new linear variable $\overline{\mathbf{y}}{:}A_1 \vee A_2$. Then we use $\vee\mathbb{L}$; we must now derive

$$x : (A_1 \to B) \wedge (A_2 \to B), \ldots; \overline{\mathbf{y}}{:}A_1 \vdash x\ \overline{\mathbf{y}} \Downarrow B \quad \text{and} \quad x : (A_1 \to B) \wedge (A_2 \to B), \ldots; \overline{\mathbf{y}}{:}A_2 \vdash x\ \overline{\mathbf{y}} \Downarrow B$$

Here, the scope of $\overline{\mathbf{y}}$ is $x\ \overline{\mathbf{y}}$, and we synthesize a type for x *twice*, once in each branch:

$$\dfrac{\ldots, y : A_1 \vee A_2; \cdot \vdash y \Uparrow A_1 \vee A_2 \qquad \dfrac{\dfrac{\ldots; \cdot \vdash x \Uparrow A_1 \to B \quad \vdots}{\ldots; \overline{\mathbf{y}}{:}A_1 \vdash x\ \overline{\mathbf{y}} \Downarrow B}\to\mathrm{E} \quad \dfrac{\ldots; \cdot \vdash x \Uparrow A_2 \to B \quad \vdots}{\ldots; \overline{\mathbf{y}}{:}A_2 \vdash x\ \overline{\mathbf{y}} \Downarrow B}\to\mathrm{E}}{\ldots; \overline{\mathbf{y}} : A_1 \vee A_2 \vdash x\ \overline{\mathbf{y}} \Downarrow B}\vee\mathbb{L}}{x : (A_1 \to B) \wedge (A_2 \to B), y : A_1 \vee A_2; \cdot \vdash x\ y \Downarrow B}\,\mathsf{direct}\mathbb{L}$$

However, when checking the translated term $\mathsf{let}\,\overline{\mathbf{x}} = x\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{y}} = y\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}}$ against B, we need to first name x as $\overline{\mathbf{x}}$, then y as $\overline{\mathbf{y}}$, then use $\vee\mathbb{L}$ to decompose the union $\overline{\mathbf{y}}{:}A_1 \vee A_2$ with subject $\mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}}$.

$$\frac{\ldots; \cdot \vdash x \Uparrow (A_1 \to B) \wedge (A_2 \to B) \qquad \ldots; \overline{\mathbf{x}}{:}(A_1 \to B) \wedge (A_2 \to B) \vdash \mathsf{let}\,\overline{\mathbf{y}} = y\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}} \Downarrow B}{x : (A_1 \to B) \wedge (A_2 \to B), y : A_1 \vee A_2; \cdot \vdash \mathsf{let}\,\overline{\mathbf{x}} = x\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{y}} = y\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}} \Downarrow B}\,\mathsf{let}$$

But we only get one chance (highlighted above) to synthesize a type for x, so we must take care when using let to name x; if we choose to synthesize $x \Uparrow A_1 \to B$ in let, we can't derive $\overline{\mathbf{x}}{:}A_1 \to B, \overline{\mathbf{y}}{:}A_2 \vdash \mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\;\overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}} \Downarrow B$ but if we choose to synthesize $x \Uparrow A_2 \to B$ we can't get $\overline{\mathbf{x}}{:}A_2 \to B, \overline{\mathbf{y}}{:}A_1 \vdash \mathsf{let}\,\overline{\mathbf{z}} = \overline{\mathbf{x}}\;\overline{\mathbf{y}}\,\mathsf{in}\,\overline{\mathbf{z}} \Downarrow B$. The only choice that works is $\Gamma(x) = (A_1 \to B) \wedge (A_2 \to B)$, since given $\overline{\mathbf{x}} \Uparrow (A_1 \to B) \wedge (A_2 \to B)$ we can synthesize $\overline{\mathbf{x}} \Uparrow A_1 \to B$ and $\overline{\mathbf{x}} \Uparrow A_2 \to B$ using $\wedge\mathrm{E}_1$ and $\wedge\mathrm{E}_2$, respectively.

In the above situation, $e' = x$ is a variable, so there is a best type C—namely $\Gamma(x)$—such that if $x \Uparrow C_1$ and $x \Uparrow C_2$ then $x \Uparrow C$, from which follows (by rules $\wedge\mathrm{E}_{1,2}$ in the example above) $x \Uparrow C_1$ and $x \Uparrow C_2$. We'll say that x has the property of *principal synthesis*. Which terms have this property? Variables do: the best type for some x is $\Gamma(x)$. On the other hand, it does not hold for many non-values: $f\ x \Uparrow A_1$ and $f\ x \Uparrow A_2$ do *not* imply $f\ x \Uparrow A_1 \wedge A_2$, since the intersection introduction rule $\wedge\mathrm{I}$ is (1) restricted to values and (2) in the checking direction. Fortunately, we don't need it for non-values: Consider $(e_1\ e_2)\ y$. Since $(e_1\ e_2)$ is not a value, y is not in evaluation position in $(e_1\ e_2)\ y$, so even in the tridirectional system, to name y we must first name $(e_1\ e_2)$. Here, the let-normal system is no more restrictive. Moreover, some values, such as pairs, are checking forms and *never* synthesize, so they do not have the principal synthesis property. But neither system binds values in checking form to linear variables.

Now, do all *values in synthesizing form* have the principal synthesis property? The only values in synthesizing form are ordinary variables x, linear variables $\overline{\mathbf{x}}$, and annotated values $(v : As)$. For x or $\overline{\mathbf{x}}$ the principal type is simply $\Gamma(x)$ or $\Delta(\overline{\mathbf{x}})$. Unfortunately, principal types do not always exist for terms of the form $(v : As)$. For example, $((\lambda x.x) : (\vdash \mathsf{unit} \to \mathsf{unit}), (\vdash \mathsf{bool} \to \mathsf{bool}))$ can synthesize $\mathsf{unit} \to \mathsf{unit}$, and it can synthesize $\mathsf{bool} \to \mathsf{bool}$, but it can't synthesize their intersection, so it has no principal type.

3.2 Slack bindings

Rather than restrict the form of annotations, we use a different kind of binding for $(v : As)$—a *slack binding* ${\sim}\overline{\mathbf{x}} = v$ where v's type is synthesized not at its binding site, but at any point up to its use (rules ∼var and let∼ in Figure 3). Wherever $\overline{\mathbf{x}}$ is in scope, we can try rule ∼var to synthesize a type A for v and replace ${\sim}\overline{\mathbf{x}} = v$ with an ordinary linear variable typing $\overline{\mathbf{x}}{:}A$. For example, $\big((\lambda x.e) : (\vdash \mathsf{int}{\to}\mathsf{int})\big)\ y$ is translated to $\mathsf{let}\,{\sim}\overline{\mathbf{x}} = \big((\lambda x.e') : (\vdash \mathsf{int}{\to}\mathsf{int})\big)\,\mathsf{in}\,\mathsf{let}\,\overline{\mathbf{y}} = y\,\mathsf{in}\,\overline{\mathbf{x}}\;\overline{\mathbf{y}}$Now, we have several chances to use ∼var to synthesize the type of $\overline{\mathbf{x}}$: just before checking $\mathsf{let}\,\overline{\mathbf{y}} = y\,\mathsf{in}\,\overline{\mathbf{x}}\;\overline{\mathbf{y}}$, or when checking $\overline{\mathbf{x}}\;\overline{\mathbf{y}}$. This is just like choosing when to apply direct$\mathbb{L}$ in the tridirectional system. If all our bindings were slack we would have put ourselves in motion to no purpose, but we'll use slack bindings for $(v : As)$ *only*. My experiments suggest that slack bindings are rare in practice (Dunfield 2007b, p. 187), and are certainly less problematic than the backtracking from intersections and unions themselves ($\wedge\mathrm{E}_{1,2}$, etc.).

4 Results

The two major results are *soundness*: if the let-normal translation of a program is well typed in the let-normal type system, the original program is well typed in the left tridirectional system—and *completeness*: if a program is well typed in the left tridirectional type system, its translation is well typed in the let-normal type system. Once these are shown, it follows from Dunfield and Pfenning (2004) that the let-normal system is sound and complete with respect to a system (Dunfield and Pfenning 2003) for which preservation and progress hold under a call-by-value semantics.

At its heart, the let-normal system merely enforces a particular pattern of linear variable introductions (via let, instead of direct$\mathbb{L}$). So it is no surprise that soundness holds. The proof is syntactic, but not too involved; see Dunfield (2007b, pp. 132–134).

Corollary (Let-Normal Soundness)**.**
If $e \hookrightarrow L + e'$ *and* $\cdot;\cdot \vdash L \,\mathsf{in}\, e' \Downarrow C$ *(let-normal system) then* $\cdot;\cdot \vdash e \Downarrow C$ *(tridirectional system).*

However, completeness—that the let-normal system is not *strictly* weaker than the tridirectional system—is involved. I give only the roughest sketch of the proof found in Dunfield (2007b, pp. 135–165). We want to show that given a well-typed term e, the let-normal translation $L \,\mathsf{in}\, e'$ where $e \hookrightarrow L + e'$ is well-typed. To be precise, given a derivation $\mathscr{D}$ deriving $\Gamma;\Delta \vdash e \Downarrow C$ in the left tridirectional system, we must construct a derivation $\Gamma;\Delta \vdash L \,\mathsf{in}\, e' \Downarrow C$ in the let-normal system, where $e \hookrightarrow L + e'$. My attempts to prove this by straightforward induction on the derivation failed: thanks to $\mathsf{direct}\mathbb{L}$, the relationship between e and $\mathscr{D}$ is complex. Nor is $L + e'$ compositional in e: for a given subterm of e there may not be a corresponding subterm of $L + e'$, because translation can insert bindings inside the translated subterm.

Instead, the completeness proof proceeds as follows:

1. *Mark* e with lets wherever $\mathsf{direct}\mathbb{L}$ is used in $\mathscr{D}$. However, if $\wedge$I or another subject-duplicating rule is used, the subderivations need not apply $\mathsf{direct}\mathbb{L}$ in the same way, resulting in distinct terms to which $\wedge$I cannot be applied. So we use step 2 inductively to obtain typing derivations for the canonical version of the subterm (the $L + e'$ from $e \hookrightarrow L + e'$), to which $\wedge$I can be applied.

   This step centres on a lemma which produces a term with a let-system typing derivation. This term might not be canonical. For example, if the original tridirectional derivation for $\lambda x.x$ didn't use $\mathsf{direct}\mathbb{L}$ at all, no let bindings are created, unlike the canonical let-normal term $\lambda x.\mathsf{let}\,\overline{\mathbf{x}} = x \,\mathsf{in}\, \overline{\mathbf{x}}$.

2. *Transform* the marked term into the canonical $L + e'$ in small steps, adding or moving one let at a time. Each small step preserves typing. We define a syntactic measure (a tuple of natural numbers) that quantifies how different a term is from $L + e'$; each let-manipulating step reduces the measure, bringing the term closer to $L + e'$. Lastly, when the measure is all zeroes, the term *is* $L + e'$.

Theorem (Let-Normal Completeness)**.**
If $\cdot;\cdot \vdash e \Downarrow C$ *(tridirectional system) and* $e \hookrightarrow L + e^*$ *then* $\cdot;\cdot \vdash L \,\mathsf{in}\, e^* \Downarrow C$ *(let-normal system).*

5 Related Work

The effects of transformation to continuation passing style on the precision of program analyses such as 0-CFA have been studied for some time (Sabry and Felleisen 1994). The effect depends on the specific details of the CPS transform and the analysis done (Damian and Danvy 2001; Palsberg and Wand 2003). The "analysis" in this work is the process of bidirectional checking/synthesis. My soundness and completeness results show that my let-normal transformation does not affect the analysis. It is not clear if this means anything for more traditional let-normal transformations and compiler analyses.

6 Conclusion

Transforming programs into a let-normal form removes a major impediment to implementing tridirectional typechecking. The system is sound *and complete* with respect to a type assignment system for intersections and unions (Dunfield and Pfenning 2003), in contrast to systems (Xi 1998) in which completeness is lost. The tridirectional rule *can* be turned into something practical. A chain of soundness results (Dunfield 2007b, p. 165) guarantees that if we run a program e whose let-normal translation typechecks in the system in this paper, it will not go wrong.

Despite "untangling" direct$\mathbb{L}$, typechecking is still very time-consuming in the worst cases, thanks to checking terms several times in $\wedge$I and backtracking in $\wedge E_{1,2}$, etc. As implementing (an extended version of) this system shows (Dunfield 2007a), bad cases do occur in practice!

Parametric polymorphism is absent, but I have extended the tridirectional system and the let-normal implementation (Dunfield 2009), and the soundness and completeness results should still hold.

The major flaw of this work is its completeness proof, which uses purely syntactic methods, is complicated, and has not been mechanized. Ideally, it would be mechanized and/or proved more simply.

Acknowledgments Many thanks to Frank Pfenning for countless discussions about this research. Thanks also to the ITRS reviewers. Most of the work was done at Carnegie Mellon University with the support of the US National Science Foundation.

References

Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *Workshop on Types in Language Design and Implementation (TLDI '05)*, pages 71–78, 2005.

Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: on the impact of the CPS transformation. Technical Report BRICS-RS-01-54, University of Aarhus, 2001.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.

Joshua Dunfield. Refined typechecking with Stardust. In *Programming Languages meets Programming Verification (PLPV '07)*, 2007a.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML Workshop (ML '09)*, 2009.

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

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures*, pages 250–266, 2003.

Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, pages 281–292, 2004.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation*, pages 237–247, 1993.

R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, 146:29–60, 1969.

Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-66, University of Edinburgh, 1988.

Jens Palsberg and Mitchell Wand. CPS transformation of flow information. *J. Functional Programming*, 13(5): 905–923, 2003.

Simon Peyton Jones and the GHC developers. Glasgow Haskell Compiler Commentary. `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscPipe`, 2006.

Benjamin C. Pierce and David N. Turner. Local type inference. In *POPL*, pages 252–265, 1998. Full version in *ACM Trans. Programming Languages and Systems*, 22(1):1–44, 2000.

John Reppy. Local CPS conversion in a direct-style compiler. In *ACM Workshop on Continuations (CW '01)*, pages 13–22, 2001.

John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993.

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

Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *Programming Language Design and Implementation*, pages 1–12, 1994.

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*, pages 181–192, 1996.

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