JFP **24** (2-3): 133–165, 2014. © Cambridge University Press 2014
doi:10.1017/S0956796813000270 First published online 20 January 2014

Elaborating intersection and union types

JANA DUNFIELD
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany
(*e-mail:* `jana.dunfield@gmail.com`)

Abstract

Designing and implementing typed programming languages is hard. Every new type system feature requires extending the metatheory and implementation, which are often complicated and fragile. To ease this process, we would like to provide general mechanisms that subsume many different features. In modern type systems, parametric polymorphism is fundamental, but intersection polymorphism has gained little traction in programming languages. Most practical intersection type systems have supported only *refinement intersections*, which increase the expressiveness of types (more precise properties can be checked) without altering the expressiveness of terms; refinement intersections can simply be erased during compilation. In contrast, *unrestricted* intersections increase the expressiveness of terms, and can be used to encode diverse language features, promising an economy of both theory and implementation. We describe a foundation for compiling unrestricted intersection and union types: an elaboration type system that generates ordinary λ-calculus terms. The key feature is a Forsythe-like merge construct. With this construct, not all reductions of the source program preserve types; however, we prove that ordinary call-by-value evaluation of the elaborated program corresponds to a type-preserving evaluation of the source program. We also describe a prototype implementation and applications of unrestricted intersections and unions: records, operator overloading, and simulating dynamic typing.

1 Introduction

In type systems, parametric polymorphism is fundamental. It enables generic programming; it supports parametric reasoning about programs. Logically, it corresponds to universal quantification.

Intersection polymorphism (the intersection type $A \wedge B$) is less well appreciated. It enables ad hoc polymorphism; it supports *irregular* generic programming, including operator overloading. Logically, it roughly corresponds to conjunction. (In our setting, this correspondence is strong, as we will see in Section 2.) Not surprisingly, then, intersection is remarkably versatile.

For both legitimate and historical reasons, intersection types have not been used as widely as parametric polymorphism. One of the legitimate reasons for the slow adoption of intersection types is that no major language has them. A restricted form of intersection, *refinement intersection*, was realized in two extensions of SML, SML-CIDRE (Davies, 2005) and Stardust (Dunfield, 2007). These type systems can express properties such as bitwise parity: after refining a type bits of bitstrings with subtypes even (an even number of ones) and odd (an odd number of ones), a bitstring concatenation function can be checked against the type

$$(\mathsf{even} * \mathsf{even} \rightarrow \mathsf{even}) \land (\mathsf{odd} * \mathsf{odd} \rightarrow \mathsf{even}) \\ \land (\mathsf{even} * \mathsf{odd} \rightarrow \mathsf{odd}) \land (\mathsf{odd} * \mathsf{even} \rightarrow \mathsf{odd})$$

which satisfies the refinement restriction: all the intersected types refine a single simple type, $\mathsf{bits} * \mathsf{bits} \rightarrow \mathsf{bits}$.

But these systems were only typecheckers. To *compile* a program required an ordinary Standard ML compiler. SML-CIDRE was explicitly limited to checking refinements of SML types, without affecting the expressiveness of terms. In contrast, Stardust could typecheck some kinds of programs that used general intersection and union types, but ineffectively: since ordinary SML compilers don't know about intersection types, such programs could never be run.

Refinement intersections and unions increase the expressiveness of otherwise more-or-less-conventional type systems, allowing more precise properties of programs to be verified through typechecking. The point is to make fewer programs pass the typechecker; for example, a concatenation function that didn't have the parity property expressed by its type would be rejected. In contrast, unrestricted intersections and unions, in cooperation with a term-level "merge" construct, increase the expressiveness of the term language. For example, given primitive operations $\texttt{Int.+} : \mathsf{int} * \mathsf{int} \rightarrow \mathsf{int}$ and $\texttt{Real.+} : \mathsf{real} * \mathsf{real} \rightarrow \mathsf{real}$, we can easily define an overloaded addition operation by writing a merge:

$$\textbf{val} + = \texttt{Int.+},, \texttt{Real.+}$$

In our type system, this function + can be checked against the type $(\mathsf{int} * \mathsf{int} \rightarrow \mathsf{int}) \land (\mathsf{real} * \mathsf{real} \rightarrow \mathsf{real})$.

In this paper, we consider unrestricted intersection and union types. Central to the approach is a method for elaborating programs with intersection and union types: elaborate intersections into products, and unions into sums. The resulting programs have no intersections and no unions, and can be compiled using conventional means—any SML compiler will do. The above definition of + is elaborated to a pair `(Int.+, Real.+)`; uses of + on ints become first projections of +, while uses on reals become second projections of +.

We present a three-phase design, based on this method, that supports one of our ultimate goals: to develop simpler compilers for full-featured type systems by encoding many features using intersections and unions.

1. An *encoding* phase that straightforwardly rewrites the program, for example, turning a multifield record type into an intersection of single-field record types, and multifield records into a "merge" of single-field records.
2. An *elaboration* phase that transforms intersections and unions into products and (disjoint) sums, and intersection and union introductions and eliminations (implicit in the source program) into their appropriate operations: tupling, projection, injection, and case analysis.
3. A *compilation* phase: a conventional compiler with no support for intersections, unions, or the features encoded by phase 1.

$$\begin{tikzcd}
 & \text{Source language} & \text{Target language} \\
 & \rightarrow, \wedge, \vee & \rightarrow, *, + \\
\text{Program} & e : A \arrow[r, hook, "\text{elaboration}"] \arrow[d, squiggly, "\substack{\text{nondeterministic}\\ \text{evaluation}\\ \text{(cbv + merge)}}"] & M : T \arrow[d, mapsto, "\substack{\text{standard}\\ \text{evaluation}\\ \text{(cbv)}}"] \\
\text{Result} & v : A \arrow[r, hook, "\text{elaboration}"] & W : T
\end{tikzcd}$$

Fig. 1. Elaboration and computation.

Contributions: Phase 2 is the main contribution of this paper. Specifically, we will:

- develop elaboration typing rules which, given a source expression e with unrestricted intersections and unions, and a "merging" construct $e_1 ,, e_2$, typecheck and transform the program into an ordinary λ-calculus term M (with sums and products);
- give a nondeterministic operational semantics ($\rightsquigarrow^*$) for source programs containing merges, in which not all reductions preserve types;
- prove a consistency (simulation) result: ordinary call-by-value evaluation ($\mapsto^*$) of the elaborated program produces a value corresponding to a value resulting from (type-preserving) reductions of the source program—that is, the diagram in Figure 1 commutes;
- describe an elaborating typechecker that, by implementing the elaboration typing rules, takes programs written in an ML-like language, with unrestricted intersection and union types, and generates Standard ML programs that can be compiled with any SML compiler.

All proofs were checked using the Twelf proof assistant (Pfenning and Schürmann, 1999; Twelf, 2012) (with the termination checker silenced for a few inductive cases, where the induction measure was nontrivial but clearly satisfied) and are available on the web (Dunfield, 2013). For convenience, the names of Twelf source files (`.elf`) are hyperlinks.

While the idea of compiling intersections to products is not new, this paper is its first full development and practical expression. An essential twist is the source-level merging construct $e_1 ,, e_2$, which embodies several computationally distinct terms, which can be checked against various parts of an intersection type, reminiscent of Forsythe (Reynolds, 1996) and (more distantly) the λ&-calculus (Castagna *et al.*, 1995). Intersections can still be introduced *without* this construct; it is required only when no single term can describe the multiple behaviors expressed by the intersection. Remarkably, this merging construct also supports union eliminations with two computationally distinct branches (unlike markers for union elimination in work such as Pierce 1991). As usual, we have no source-level intersection eliminations and no source-level union introductions; elaboration puts all needed projections and injections into the target program.

Contents: In Section 2, we give some brief background on intersection types, discuss their introduction and elimination rules, introduce and discuss the merge construct, and compare intersection types to product types. Section 3 gives background on union types, discusses *their* introduction and elimination rules, and shows how the merge construct is also useful for them.

Section 4 has the details of the source language and its (unusual) operational semantics, and describes a nonelaborating type system including subsumption. Section 5 presents the target language and its (entirely standard) typing and operational semantics. Section 6 gives the elaboration typing rules, and proves several key results relating source typing, elaboration typing, the source operational semantics, and the target operational semantics.

Section 7 discusses a major caveat: the approach, at least in its present form, lacks the theoretically and practically important property of coherence, because the meaning of a target program depends on the choice of elaboration typing derivation.

Section 8 shows encodings of type system features into intersections and unions, with examples that are successfully elaborated by our prototype implementation (Section 9). Related work is discussed in Section 10, and Section 11 concludes.

Previous version: An earlier version of this work (Dunfield, 2012) appeared at the International Conference on Functional Programming (ICFP). The technical details are essentially unchanged, except for a simpler Lemma 9 (the old lemma is an immediate corollary of the new one), but several sections have been expanded and clarified, particularly the discussion of bidirectional typechecking; also, this version includes a link to the implementation.

2 Intersection types

What is an intersection type? The simplistic answer is that, supposing that types describe sets of values, $A \wedge B$ describes the intersection of the sets of values of A and B. That is, $v : A \wedge B$ if $v : A$ and $v : B$.

Less simplistically, the name has been used for substantially different type constructors, though all have a conjunctive flavor. The intersection type in this paper is commutative ($A \wedge B = B \wedge A$) and idempotent ($A \wedge A = A$), following several of the seminal papers on intersection types (Pottinger, 1980; Coppo *et al.*, 1981), and more recent work with refinement intersections (Freeman and Pfenning, 1991; Davies and Pfenning, 2000; Dunfield and Pfenning, 2003). Other lines of research have worked with nonlinear and/or ordered intersections, e.g., Kfoury and Wells (2004), which seem less directly applicable to practical type systems (Møller Neergaard and Mairson, 2004).

For this paper, then: What is a commutative and idempotent intersection type?

One approach to this question is through the Curry–Howard correspondence. Naively, intersection should correspond to logical conjunction—but products correspond to logical conjunction, and intersections are not products, as is evident from comparing the standard[1] introduction and elimination rules for intersection to the (utterly standard) rules for

[1] For impure call-by-value languages like ML, $\wedge$I ordinarily needs to be restricted to type a value v, for reasons analogous to the value restriction on parametric polymorphism (Davies and Pfenning, 2000). Our setting, however, is not ordinary: the technique of elaboration makes the more permissive rule safe, though user-unfriendly. See Section 6.5.

product. (Throughout this paper, k is existentially quantified over $\{1,2\}$; technically, and in the Twelf formulation, we have two rules $\wedge E_1$ and $\wedge E_2$, etc.)

$$\frac{e : A_1 \qquad e : A_2}{e : A_1 \wedge A_2}\,\wedge I \qquad\qquad \frac{e : A_1 \wedge A_2}{e : A_k}\,\wedge E_k$$

$$\frac{e_1 : A_1 \qquad e_2 : A_2}{(e_1\,,\, e_2) : A_1 * A_2}\,{*I} \qquad\qquad \frac{e : A_1 * A_2}{\textbf{proj}_k\; e : A_k}\,{*E_k}$$

Here, $\wedge I$ types a single term e which inhabits type A_1 *and* type A_2: via Curry–Howard, this means that a single proof term serves as witness to two propositions (the interpretations of A_1 and A_2). On the other hand, in $*I$ two separate terms e_1 and e_2 witness the propositions corresponding to A_1 and A_2. This difference was suggested by Pottinger (1980), and made concrete when Hindley (1984) showed that intersection (of the form described by Coppo *et al.* 1981 and Pottinger 1980) cannot correspond to conjunction because the following type, the intersection of the types of the I and S combinators, is uninhabited:

$$(A \rightarrow A) \wedge \underbrace{\big((A{\rightarrow}B{\rightarrow}C) \rightarrow (A{\rightarrow}B) \rightarrow A \rightarrow C\big)}_{\text{“D”}}$$

yet the prospectively corresponding proposition is provable in intuitionistic logic:

$$(A \supset A) \text{ and } \big((A{\supset}B{\supset}C) \supset (A{\supset}B) \supset A \supset C\big) \qquad (*)$$

Hindley notes that every term of type $A \rightarrow A$ is β-equivalent to $e_1 = \lambda x.\, x$, the I combinator, and every term of type D is β-equivalent to $e_2 = \lambda x.\, \lambda y.\, \lambda z.\, x\, z\, (y\, z)$, the S combinator. Any term e of type $(A \rightarrow A) \wedge D$ must therefore have two normal forms, e_1 and e_2, which is impossible.

But that impossibility holds for the *usual* λ-terms. Suppose we add a *merge* construct $e_1 ,, e_2$ that, quite brazenly, can step to two different things: $e_1 ,, e_2 \mapsto e_1$ and $e_1 ,, e_2 \mapsto e_2$. Its typing rule chooses one subterm and ignores the other:

$$\frac{e_k : A}{e_1 ,, e_2 : A}\,\mathsf{merge}_k$$

In combination with $\wedge I$, the merge_k rule allows two distinct implementations e_1 and e_2, one for each of the components A_1 and A_2 of the intersection:

$$\frac{\dfrac{e_1 : A_1}{e_1 ,, e_2 : A_1}\,\mathsf{merge}_1 \qquad \dfrac{e_2 : A_2}{e_1 ,, e_2 : A_2}\,\mathsf{merge}_2}{e_1 ,, e_2 : A_1 \wedge A_2}\,\wedge I$$

Now $(A \rightarrow A) \wedge D$ *is* inhabited:

$$e_1 ,, e_2 : (A \rightarrow A) \wedge D$$

With this construct, the “naive” hope that intersection corresponds to conjunction is realized through elaboration: we can elaborate $e_1 ,, e_2$ to $(e_1\,,\, e_2)$, a term of type $(A \rightarrow A) * D$, which does correspond to the proposition (*). Inhabitation and provability again correspond—because we have replaced the seemingly mysterious intersections with simple products.

For source expressions, intersection still has several properties that set it apart from product. Unlike product, it has no elimination form. It also lacks an explicit introduction form; $\wedge$I is the only intro rule for $\wedge$. While the primary purpose of merge_k is to derive the premises of $\wedge$I, the merge_k rule makes no mention of intersection (or any other type constructor).

Pottinger (1980) presents intersection $A \mathbin{\widehat{\&}} B$ as a proposition with some evidence of A that is also evidence of B—unlike $A \mathbin{\&} B$, corresponding to $A * B$, which has two separate pieces of evidence for A and for B. In our system, though, $e_1 ,, e_2$ is a single term that provides evidence for A and B, so it is technically consistent with this view of intersection, but not necessarily consistent in spirit (since e_1 and e_2 can be very different from each other).

3 Union types

Having discussed intersection types, we can describe union types as intersections' dual: if $v : A_1 \vee A_2$ then either $v : A_1$ or $v : A_2$ (perhaps both). This duality shows itself in several ways.

For union $\vee$, introduction is straightforward, as elimination was straightforward for $\wedge$ (again, k is either 1 or 2):

$$\frac{\Gamma \vdash e : A_k}{\Gamma \vdash e : A_1 \vee A_2}\ \vee\mathrm{I}_k$$

Here, the term e inhabits both A_k and $A_1 \vee A_2$. Thus, we have one proof term that witnesses two propositions, in contrast to the usual introduction rule for sums where e is evidence of A_k only, and $\mathbf{inj}_k\ e$ is evidence of $A_1 + A_2$:

$$\frac{\Gamma \vdash e : A_k}{\Gamma \vdash \mathbf{inj}_k\ e : A_1 + A_2}\ +\mathrm{I}_k$$

This corresponds to logical disjunction, with an explicit or-introduction in the proof term.

For the elimination rule, first consider the usual elimination rule for sums:

$$\frac{\Gamma \vdash e_0 : A_1 + A_2 \qquad \begin{array}{c}\Gamma, x : A_1 \vdash e_1 : C \\ \Gamma, x : A_2 \vdash e_2 : C\end{array}}{\Gamma \vdash (\mathbf{case}\ e_0\ \mathbf{of}\ \mathbf{inj}_1\ x \Rightarrow e_1 \mid \mathbf{inj}_2\ x \Rightarrow e_2) : C}\ +\mathrm{E}$$

By analogy with the rules for intersection and union given above, a single term e should serve as evidence in both branches, instead of two pieces of evidence e_1 and e_2. Moreover, since we introduce (and eliminate) intersection without an explicit syntactic form, we expect to eliminate union without an explicit syntactic form. So, the rule should look something like

$$\frac{\Gamma \vdash e_0 : A_1 \vee A_2 \qquad \begin{array}{c}\Gamma, x : A_1 \vdash e : C \\ \Gamma, x : A_2 \vdash e : C\end{array}}{\Gamma \vdash [e_0/x]e : C}$$

The subject of the conclusion is some term with (zero or more) occurrences of e_0; the term e in the premises is the same, but with x in place of e_0. (We write $[e_0/x]e$ for the substitution of e_0 for x in e.) We can view e as taking x as a parameter, where x is evidence of either A_1 or of A_2; in either case, e is evidence of C.

However, the rule above is unsound in many settings (see the discussion of call-by-value, below); we use a rule that is sound for call-by-value, and acceptably strong:

$$\frac{\Gamma \vdash e_0 : A_1 \vee A_2 \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash \mathcal{E}[x_1] : C \\ \Gamma, x_2 : A_2 \vdash \mathcal{E}[x_2] : C\end{array}}{\Gamma \vdash \mathcal{E}[e_0] : C}\,\vee\mathrm{E}$$

This rule types an expression $\mathcal{E}[e_0]$—an evaluation context $\mathcal{E}$ where e_0 occurs in an evaluation position—where e_0 has the union type $A_1 \vee A_2$. During evaluation, e_0 will be some value v_0 such that either $v_0 : A_1$ or $v_0 : A_2$. In the former case, the premise $x_1 : A_1 \vdash \mathcal{E}[x_1] : C$ tells us that substituting v_0 for x_1 gives a well-typed expression $\mathcal{E}[v_0]$. Similarly, the premise $x_2 : A_2 \vdash \mathcal{E}[x_2] : C$ tells us we can safely substitute v_0 for x_2.

The restriction to a single occurrence of e_0 in an evaluation position is needed for soundness in many settings—generally, in any operational semantics in which e_0 might step to different expressions. One simple example is a function $f : (A_1 \rightarrow A_1 \rightarrow C) \wedge (A_2 \rightarrow A_2 \rightarrow C)$ and expression $e_0 : A_1 \vee A_2$, where e_0 mutates a reference cell that has type $\mathbf{ref}\ (A_1 \vee A_2)$, then returns the new stored value. The application $f\ e_0\ e_0$ would be well-typed by a rule allowing multiple occurrences of e_0, but unsound: the first e_0 could evaluate to some value $v_1 : A_1$ and the second e_0 to some $v_2 : A_2$, yielding the ill-typed application $f\ v_1\ v_2$.

In this paper, we are interested only in call-by-value languages. The choice of evaluation strategy does affect the type system, but some variants of the union elimination rule are unsound under both call-by-value and call-by-name. Barbanera *et al.* (1995) discuss such a rule—and define an unusual "parallel reduction" semantics for which it *is* sound. For further discussion of this rule, see Dunfield and Pfenning (2003). Finally, note that the evaluation context $\mathcal{E}$ need not be unique, which creates some difficulties for practical typechecking (Dunfield, 2011).

We saw in Section 2 that, in the usual λ-calculus, $\wedge$ does not correspond to conjunction; in particular, no λ-term behaves like both the I and S combinators, so the intersection $(A \rightarrow A) \wedge D$ (where D is the type of S) is uninhabited. In our setting, though, $(A \rightarrow A) \wedge D$ *is* inhabited, by the merge of I and S.

Something similar comes up when eliminating unions. Without the merge construct, certain instances of union types can't be usefully eliminated. Consider a list whose elements have type $\mathsf{int} \vee \mathsf{string}$. Introducing those unions to create the list is easy enough: use $\vee I_1$ for the ints and $\vee I_2$ for the strings. Now suppose we want to print a list element $x : \mathsf{int} \vee \mathsf{string}$, converting the ints to their string representation and leaving the strings alone. To do this, we need a merge; for example, given a function $g : (\mathsf{int} \rightarrow \mathsf{string}) \wedge (\mathsf{string} \rightarrow \mathsf{string})$ whose body contains a merge, use rule $\vee$E on $g\ x$ with $\mathcal{E} = g\ []$ and $e_0 = x$:

$$\frac{\Gamma \vdash x : \mathsf{int} \vee \mathsf{string} \qquad \begin{array}{c}\Gamma, x_1 : \mathsf{int} \vdash g\ x_1 : \mathsf{string} \\ \Gamma, x_2 : \mathsf{string} \vdash g\ x_2 : \mathsf{string}\end{array}}{\Gamma \vdash g\ x : \mathsf{string}}\,\vee\mathrm{E}$$

Source types	$A, B, C ::= \top \mid A \to B \mid A \wedge B \mid A \vee B$
Typing contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$
Source expressions	$e ::= x \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid \textbf{fix}\, x.\, e \mid e_1 ,, e_2$
Source values	$v ::= x \mid () \mid \lambda x.\, e \mid v_1 ,, v_2$
Evaluation contexts	$\mathcal{E} ::= [] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid \mathcal{E} ,, e \mid e ,, \mathcal{E}$

Fig. 2. Syntax of source types, contexts, and expressions.

Because of $\vee$E, typing is not always preserved by η-reduction. Thus, we must sometimes η-expand, as in the coercion for the subtyping rule $\vee$L$\leqslant$ (see Section 4.4 and the proof of Theorem 1) and in one of our examples (see the discussion in Section 8.3).

Like intersections, unions can be tamed by elaboration. Instead of products, we elaborate unions to products' dual, sums (*tagged* unions). Uses of $\vee$I$_1$ and $\vee$I$_2$ become left and right injections into a sum type; uses of $\vee$E become ordinary case expressions.

4 Source language

4.1 Source syntax

The source language expressions e are standard, except for the feature central to our approach, the merge $e_1 ,, e_2$. The types A, B, C are: a "top" type $\top$, whose values carry no information, and which we will elaborate to unit; the usual function space $A \to B$; intersection $A \wedge B$; and union $A \vee B$. Note that $\top$ can be viewed as a 0-ary intersection. Values v are standard, except that a merge of values $v_1 ,, v_2$ is considered a value even though it can step! But the step it takes is pure, in the sense that even if we incorporated effects such as mutable references, it would not interact with them.

As usual, we follow Barendregt's convention of automatically renaming bound variables, and use a standard capture-avoiding substitution $[e'/x]e$ (e' substituted for x in e). In typing contexts Γ, we assume that variables are not declared more than once. Finally, we treat contexts as ordered lists, though this is not required in the setting of this paper.

4.2 Source operational semantics

The source language operational semantics (Figure 3) is standard, with (left-to-right) call-by-value function application and a fixed-point expression, except for the merge construct. This peculiar animal is a descendant of "demonic choice" (often written $\oplus$): by the 'step/unmerge left' and 'step/unmerge right' rules, $e_1 ,, e_2$ can step to either e_1 or e_2. Adding to its misbehaviours, it permits stepping within itself, via 'step/merge1' and 'step/merge2'—note that in 'step/merge2', we don't require e_1 to be a value. Worst of all, it can appear by spontaneous fission: 'step/split' turns any expression e into a merge of two copies of e.

The merge construct makes our source language operational semantics interesting. It also makes it unrealistic: $\rightsquigarrow$-reduction does not preserve types. For type preservation to hold, the operational semantics would need access to the typing derivation. Even worse, since the typing rule for merges ignores the unused part of the merge, $\rightsquigarrow$-reduction can

$\boxed{e \leadsto e'}$ Source expression e steps to e' $\boxed{\texttt{step E E' in step.elf}}$

$$\frac{e_1 \leadsto e_1'}{e_1\,e_2 \leadsto e_1'\,e_2}\;\text{step/app1} \qquad \frac{e_2 \leadsto e_2'}{v_1\,e_2 \leadsto v_1\,e_2'}\;\text{step/app2}$$

$$\frac{}{(\lambda x.\,e)v \leadsto [v/x]e}\;\text{step/beta} \qquad \frac{}{\textbf{fix}\ x.\,e \leadsto [(\textbf{fix}\ x.\,e)/x]e}\;\text{step/fix}$$

$$\frac{}{e_1 \,,,\, e_2 \leadsto e_1}\;\text{step/unmerge left} \qquad \frac{}{e_1 \,,,\, e_2 \leadsto e_2}\;\text{step/unmerge right}$$

$$\frac{e_1 \leadsto e_1'}{e_1 \,,,\, e_2 \leadsto e_1' \,,,\, e_2}\;\text{step/merge1} \qquad \frac{e_2 \leadsto e_2'}{e_1 \,,,\, e_2 \leadsto e_1 \,,,\, e_2'}\;\text{step/merge2}$$

$$\frac{}{e \leadsto e \,,,\, e}\;\text{step/split}$$

Fig. 3. Source language operational semantics: call-by-value + merge construct.

produce expressions that have no type at all—or, if the unused part of the merge is ill-formed, are not even closed!

The point of the source operational semantics is not to directly model computation; rather, it is a basis for checking that the elaborated program (whose operational semantics is perfectly standard) makes sense. We will show in Section 6 that, if the result M of elaborating e can step to some M', then we can step $e \leadsto^* e'$ where e' elaborates to M'. The peculiar rule 'step/split' is used in the proof of Lemma 11, in the case for $\wedge$I; introducing a merge allows us to compose the result of applying the induction hypothesis to each subderivation.

4.3 (Source) subtyping

Suppose we want to pass a function $f : A \to C$ to a function $g : ((A \wedge B) \to C) \to D$. This should be possible, since f requires only that its argument have type A; in all calls from g the argument to f will also have type B, but f won't mind. With only the rules discussed so far, however, the application $g\ f$ is not well typed: we can't eliminate the intersection $A \wedge B$ under the arrow in $(A \wedge B) \to C$. For flexibility, we'll incorporate a subtyping system that can conclude, for example, $A \to C \leqslant (A \wedge B) \to C$.

The logic of the subtyping rules (Figure 4, top) is taken straight from Dunfield and Pfenning (2003). Roughly, $A \leqslant B$ is sound if every value of type A can be treated as having type B. Under a subset interpretation, this would mean that $A \leqslant B$ is justified if the set of A-values is a subset of the set of B-values. For example, the rule $\wedge$R$\leqslant$, interpreted set-theoretically, says that if $A \subseteq B_1$ and $A \subseteq B_2$ then $A \subseteq (B_1 \cap B_2)$. We can also take the perspective of the sequent calculus (Gentzen, 1969), and read $A \leqslant B$ as $A \vdash B$: The left and right subtyping rules for intersection correspond to the left and right rules for conjunction in the sequent calculus, but with a single antecedent and succedent. Likewise, the subtyping rules for union correspond to the rules for disjunction in the sequent calculus.

Our rules are simple and *orthogonal*: the subtyping behavior of each type constructor can be understood independently, because no rule mentions two different constructors. Hence, we have no distributivity properties, such as that of $\wedge$ over $\to$, or of $\wedge$ and $\vee$ over

$\boxed{A \le B ::: e}$ Source type A is a subtype of source type B, with coercion e of type $\cdot \vdash e : A \to B$ $\boxed{\textit{sub A B Coe CoeTyping} \text{ in } \textit{typeof+sub.elf}}$

$$\frac{B_1 \le A_1 ::: e \qquad A_2 \le B_2 ::: e'}{A_1 \to A_2 \le B_1 \to B_2 ::: \lambda f.\lambda x.\, e'\,(f\,(e\,x))}\to\le \qquad \frac{}{A \le \top ::: \lambda x.\,()}\top R\le$$

$$\frac{A_k \le B ::: e}{A_1 \wedge A_2 \le B ::: e}\wedge L_k\le \qquad \frac{A \le B_1 ::: e_1 \qquad A \le B_2 ::: e_2}{A \le B_1 \wedge B_2 ::: e_1 ,, e_2}\wedge R\le$$

$$\frac{A_1 \le B ::: e_1 \qquad A_2 \le B ::: e_2}{A_1 \vee A_2 \le B ::: \lambda x.\,(\lambda y.\, e_1\, y ,, e_2\, y)\, x}\vee L\le \qquad \frac{A \le B_k ::: e}{A \le B_1 \vee B_2 ::: e}\vee R_k\le$$

$\boxed{\Gamma \vdash e : A}$ Source expression e has source type A $\boxed{\textit{typeof+sub E A} \text{ in } \textit{typeof+sub.elf}}$

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x : A}\text{var} \qquad \frac{\Gamma \vdash e_k : A}{\Gamma \vdash e_1 ,, e_2 : A}\text{merge}_k \qquad \frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \mathbf{fix}\, x.\, e : A}\text{fix} \qquad \frac{}{\Gamma \vdash v : \top}\top I$$

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

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

$$\frac{\Gamma \vdash e_0 : A \qquad \Gamma, x : A \vdash \mathscr{E}[x] : C}{\Gamma \vdash \mathscr{E}[e_0] : C}\text{direct}$$

$$\frac{\Gamma \vdash e : A_k}{\Gamma \vdash e : A_1 \vee A_2}\vee I_k \qquad \frac{\Gamma \vdash e_0 : A_1 \vee A_2 \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash \mathscr{E}[x_1] : C \\ \Gamma, x_2 : A_2 \vdash \mathscr{E}[x_2] : C\end{array}}{\Gamma \vdash \mathscr{E}[e_0] : C}\vee E$$

$$\frac{\Gamma \vdash e : A \qquad A \le B ::: e_{\text{coerce}}}{\Gamma \vdash e : B}\text{sub}$$

Fig. 4. Source type system, with subsumption, nonelaborating.

each other. Including distributivity of $\wedge$ over $\to$ is problematic, for similar reasons as the value restriction on $\wedge$-introduction; see Davies and Pfenning (2000) and Section 6.5 below. Distributivity of $\wedge$ and $\vee$ over each other appears safe, but would defeat orthogonality. Since our rules do not capture all sound subtyping relationships, they are incomplete. (This very syntactic approach stands in marked contrast to the *semantic subtyping* approach of Frisch *et al.* (2008), which aims to capture *all* sound subtypings.)

It is easy to show that subtyping is reflexive and transitive:

Lemma.
Given a type A, *there exists* e *such that* $A \leqslant A ::: e$.

Proof
By structural induction on A; see *sub-refl.elf*. □

Lemma.
If $A \leqslant B ::: e_{AB}$ *and* $B \leqslant C ::: e_{BC}$ *then there exists* e_{AC} *such that* $A \leqslant C ::: e_{AC}$.

Proof
By simultaneous induction on the given derivations; see `sub-trans.elf`. □

Note that building transitivity into the structure of the rules makes it easy to derive an algorithm; an explicit transitivity rule would have premises $A \leqslant B$ and $B \leqslant C$, which involve an intermediate type B that does not appear in the conclusion $A \leqslant C$.

Having said all that, the subsequent theoretical development is easier without subtyping. So we will show (Theorem 1) that, given a typing derivation that uses subtyping (through the usual subsumption rule), we can always construct a source expression of the same type that never applies the subsumption rule. This new expression will be the same as the original one, with a few additional coercions. For the example above, we essentially η-expand $g\ f$ to $g\ (\lambda x.\, f\ x)$, which lets us apply $\wedge E_1$ to $x : A \wedge B$. More generally, adding coercions $\beta\eta$-expands the expression, "articulating" the type structure and making the subsumption rule unnecessary. All the coercions are identities except for rule $\top R\leqslant$, which can replace any value used at type unit with the "canonical" unit value $()$.

This is a long-standing technique for systems with subtyping over intersection types; Barendregt *et al.* (1983) used it in a completeness argument, showing that no typings are lost when the subsumption rule is replaced by a $\beta\eta$-expansion rule (their Lemma 4.2).

Note that the coercion in rule $\vee L\leqslant$ is itself η-expanded to allow $\vee E$ to eliminate the union in the type of x, since the subexpression of union type must be in evaluation position.

4.4 Source typing

The source typing rules (Figure 4) are either standard or have already been discussed in Sections 2 and 3, except for $\top I$ and direct.

The $\top I$ rule says that any value can be given type $\top$. It types *any* value, not just the unit expression $()$—even though, given rule sub, we could get the same effect with a version of $\top I$ that typed only $()$ (and prove exactly the same results). However, the more general $\top I$ more closely resembles $\wedge I$, emphasizing that $\top$ is essentially a 0-ary version of $\wedge$.[2]

The direct rule was introduced and justified in Dunfield and Pfenning (2003, 2004). It is a 1-ary version of $\vee E$, a sort of cut: it allows us to replace a derivation of $\mathcal{E}[e_0] : C$ that contains a subderivation of $e_0 : A$ by a derivation of $e_0 : A$, along with a derivation of $\mathcal{E}[x] : C$ that assumes $x : A$. Curiously, in this system of rules, direct is admissible: given $e_0 : A$, use $\vee I_1$ or $\vee I_2$ to conclude $e_0 : A \vee A$, then use two copies of the derivation $x : A \vdash \mathcal{E}[x] : C$ in the premises of $\vee E$ (α-converting x as needed). So why include it? Typing using these rules is undecidable; our implementation uses a *bidirectional* version of these rules in which typechecking is decidable given a few annotations (Dunfield and Pfenning, 2004). That bidirectional system (Section 9.1) has two judgment forms, checking and synthesis, and in that system direct is *not* admissible.

Remark. Theorem 1, and all subsequent theorems, are proved only for expressions that are closed under the appropriate context Γ. While rule merge_k does not explicitly check

[2] In $\wedge I$, a value restriction is mandatory in only some settings (Section 6.5). But we cannot let $\top I$ give type $\top$ to expressions that are not values: we will elaborate such values to the target term $()$, but some source expressions never step to values, which would break the correspondence between the source and target semantics. Specifically, Lemma 11 would fail.

$$\begin{array}{rrll}
\text{Target types} & \mathsf{T} & ::= & \mathsf{unit} \mid \mathsf{T} \to \mathsf{T} \mid \mathsf{T} * \mathsf{T} \mid \mathsf{T} + \mathsf{T} \\
\text{Typing contexts} & \mathsf{G} & ::= & \cdot \mid \mathsf{G}, x : \mathsf{T} \\
\text{Target terms} & M, N & ::= & x \mid () \mid \lambda x.\, M \mid M\, N \mid \mathbf{fix}\; x.\, M \\
& & & \mid (M_1\,,\, M_2) \mid \mathbf{proj}_k\; M \\
& & & \mid \mathbf{inj}_k\; M \mid \mathbf{case}\; M\; \mathbf{of}\; \mathbf{inj}_1\; x_1 \Rightarrow N_1 \\
& & & \qquad\qquad\qquad\qquad \mid \mathbf{inj}_2\; x_2 \Rightarrow N_2 \\
\text{Target values} & W & ::= & x \mid () \mid \lambda x.\, M \\
& & & \mid (W_1\,,\, W_2) \\
& & & \mid \mathbf{inj}_k\; W
\end{array}$$

Fig. 5. Target types and terms.

that the unexamined subexpression be closed, our implementation does perform this check (when it parses the program). Since Twelf does not support proofs about objects with unknown variables, the implementation and the proof are in harmony.

Theorem 1 (*Coercion*)
If $\mathcal{D}$ derives $\Gamma \vdash e : B$ then there exists an e' such that $\mathcal{D}'$ derives $\Gamma \vdash e' : B$, where $\mathcal{D}'$ never uses rule sub.

Proof
By induction on $\mathcal{D}$. The interesting cases are for sub and $\vee$E. In the case for sub with $A \leqslant B$, we show that when the coercion e_{coerce}—which always has the form $\lambda x.\, e_0$—is applied to an expression of type A, we get an expression of type B. For example, for $\wedge L_1 \leqslant$ we use $\wedge E_1$. This shows that $e' = (\lambda x.\, e_0)\; e$ has type B.

For $\vee$E, the premises typing $\mathcal{E}[x_k]$ might "separate," say if the first includes subsumption (yielding the same $\mathcal{E}[x_1]$) and the second doesn't. Furthermore, inserting coercions could break evaluation positions: given $\mathcal{E} = f\;[]$, replacing f with an application $(e_{\text{coerce}}\; f)$ means that $[]$ is no longer in evaluation position. The solution is to let $e' = (\lambda y.\, e'_1 ,, e'_2)\; e'_0$ where e'_0 comes from applying the induction hypothesis to the derivation of $\Gamma \vdash e_0 : A_1 \vee A_2$, and e'_1 and e'_2 come from applying the induction hypothesis to the other two premises. Now e'_0 *is* in evaluation position, because it follows a value $(\lambda y.\, e'_1 ,, e'_2)$; the merge_k typing rule will choose the correct branch.

For details, see *coerce.elf*. We actually encode the typings for e_{coerce} as hypothetical derivations in the subtyping judgment itself (*typeof+sub.elf*), making the sub case here trivial. □

5 Target language

Our target language is just the simply-typed call-by-value λ-calculus extended with fixed-point expressions, products, and sums.

5.1 Target syntax

The target types and terms (Figure 5) are completely standard.

$\boxed{G \vdash M : T}$ Target term M has target type T $\boxed{\texttt{typeoftm } M\ T \texttt{ in typeoftm.elf}}$

$$\frac{}{G_1, x : T, G_2 \vdash x : T}\ \text{typeoftm/var} \qquad \frac{G, x : T \vdash M : T}{G \vdash \textbf{fix}\, x.\, M : T}\ \text{typeoftm/fix} \qquad \frac{}{G \vdash () : \mathsf{unit}}\ \text{typeoftm/unitintro}$$

$$\frac{G, x : T_1 \vdash M : T_2}{G \vdash \lambda x.\, M : (T_1 \to T_2)}\ \text{typeoftm/arrintro} \qquad \frac{G \vdash M_1 : T \to T' \qquad G \vdash M_2 : T}{G \vdash M_1\, M_2 : T'}\ \text{typeoftm/arrelim}$$

$$\frac{G \vdash M_1 : T_1 \qquad G \vdash M_2 : T_2}{G \vdash (M_1\, ,\, M_2) : (T_1 * T_2)}\ \text{typeoftm/prodintro} \qquad \frac{G \vdash M : (T_1 * T_2)}{G \vdash (\textbf{proj}_k\, M) : T_k}\ \text{typeoftm/prodelim}_k$$

$$\frac{G \vdash M : T_k}{G \vdash (\textbf{inj}_k\, M) : (T_1 + T_2)}\ \text{typeoftm/sumintro}_k \qquad \frac{G \vdash M : T_1 + T_2 \qquad \begin{array}{l} G, x_1 : T_1 \vdash N_1 : T \\ G, x_2 : T_2 \vdash N_2 : T \end{array}}{G \vdash (\textbf{case}\ M\ \textbf{of inj}_1\, x_1 \Rightarrow N_1 \mid \textbf{inj}_2\, x_2 \Rightarrow N_2) : T}\ \text{typeoftm/sumelim}$$

Fig. 6. Target type system with functions, products, and sums.

5.2 *Target typing*

The typing rules for the target language (Figure 6) lack any form of subtyping, and are completely standard.

5.3 *Target operational semantics*

The operational semantics $M \mapsto M'$, read M steps to M', is also standard; functions are call-by-value and products are strict. As usual, we write $M \mapsto^* M'$ for a sequence of zero or more $\mapsto$-steps. Naturally, a type safety result and a determinism result hold. Note that the main results of the paper don't depend on these theorems: their purpose is to reassure us that we have defined the target semantics correctly.

Theorem 2 (*Target Type Safety*)
If $\cdot \vdash M : T$ then either M is a value, or $M \mapsto M'$ and $\cdot \vdash M' : T$.

Proof
By induction on the given derivation, using a few standard lemmas; see *tm-safety.elf*. (The necessary substitution lemma comes for free in Twelf.) □

Theorem 3 (*Determinism of* $\mapsto$)
If $M \mapsto N_1$ and $M \mapsto N_2$ then $N_1 = N_2$ (up to α-conversion).

Proof
By simultaneous induction. See *tm-deterministic* in *tm-safety.elf*. □

6 Elaboration typing

We elaborate well-typed source expressions e into target terms M. The source expressions, which include a "merge" construct $e_1\,,,\,e_2$, are typed with intersections and unions, but the result of elaboration is completely standard and can be typed with just unit, $\to$, $*$ and $+$.

$\boxed{M \mapsto M'}$ Target term M steps to M' $\boxed{\texttt{steptm M M' in steptm.elf}}$

$$\frac{M_1 \mapsto M_1'}{M_1 M_2 \mapsto M_1' M_2} \qquad \frac{M_2 \mapsto M_2'}{W_1 M_2 \mapsto W_1 M_2'}$$

$$\frac{}{(\lambda x.\, M) W \mapsto [W/x]M} \qquad \frac{}{\mathbf{fix}\; x.\, M \mapsto [(\mathbf{fix}\; x.\, M)/x]M}$$

$$\frac{M \mapsto M'}{\mathbf{proj}_k\; M' \mapsto \mathbf{proj}_k\; M'} \qquad \frac{}{\mathbf{proj}_k\; (W_1\,,\, W_2) \mapsto W_k}$$

$$\frac{M_1 \mapsto M_1'}{(M_1\,,\, M_2) \mapsto (M_1'\,,\, M_2)} \qquad \frac{M_2 \mapsto M_2'}{(W_1\,,\, M_2) \mapsto (W_1\,,\, M_2')}$$

$$\frac{M \mapsto M'}{\mathbf{inj}_k\; M \mapsto \mathbf{inj}_k\; M'} \qquad \frac{M \mapsto M'}{\mathbf{case}\; M\; \mathbf{of}\; MS \mapsto \mathbf{case}\; M'\; \mathbf{of}\; MS}$$

$$\frac{}{\mathbf{case}\; \mathbf{inj}_k\; W\; \mathbf{of}\; \mathbf{inj}_1\; x_1 \Rightarrow N_1 \;\|\; \mathbf{inj}_2\; x_2 \Rightarrow N_2 \;\mapsto\; [W/x_k]N_k}$$

Fig. 7. Target language operational semantics: call-by-value + products + sums.

The elaboration judgment $\Gamma \vdash e : A \hookrightarrow M$ is read "under assumptions Γ, source expression e has type A and elaborates to target term M." While not written explicitly in the judgment, the elaboration rules ensure that M has type $|A|$, the *type translation* of A (Figure 9). For example, $|\top \wedge (\top \rightarrow \top)| = \mathsf{unit} * (\mathsf{unit} \rightarrow \mathsf{unit})$.

To simplify the technical development, the elaboration rules work only for source expressions that can be typed without using the subsumption rule sub (Figure 4). Such source expressions can always be produced (Theorem 1, above).

In the rest of this section, we discuss the elaboration rules and prove related properties:

6.1 connects elaboration, source typing, and target typing;
6.2 gives lemmas useful for showing that target computations correspond to source computations;
6.3 states and proves that correspondence (*consistency*, Thm. 13);
6.4 summarizes the metatheory through two important corollaries of our theorems.
6.5 discusses whether we need a value restriction on $\wedge$I.

6.1 Connecting elaboration and typing

Equivalence of elaboration and source typing: The nonelaborating type assignment system of Figure 4, minus sub, can be read off from the elaboration rules in Figure 8: simply drop the $\hookrightarrow \ldots$ part of the judgment. Consequently, given $e : A \hookrightarrow M$ we can always derive $e : A$:

Theorem 4
If $\Gamma \vdash e : A \hookrightarrow M$ then $\Gamma \vdash e : A$ (without using rule sub).

Proof
By induction on the given derivation; see `typeof-erase` in `typeof-elab.elf`. □

$\boxed{\Gamma \vdash e : A \hookrightarrow M}$ Source expression e has source type A and elaborates to target term M (of type $|A|$) $\boxed{\textit{elab } E\ A\ M \text{ in } \textit{elab.elf}}$

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x : A \hookrightarrow x}\ \mathsf{var} \qquad \frac{\Gamma \vdash e_k : A \hookrightarrow M}{\Gamma \vdash e_1 ,, e_2 : A \hookrightarrow M}\ \mathsf{merge}_k$$

$$\frac{\Gamma, x : A \vdash e : A \hookrightarrow M}{\Gamma \vdash \mathbf{fix}\ x.\, e : A \hookrightarrow \mathbf{fix}\ x.\, M}\ \mathsf{fix} \qquad \frac{}{\Gamma \vdash v : \top \hookrightarrow ()}\ \top\mathrm{I}$$

$$\frac{\Gamma, x : A \vdash e : B \hookrightarrow M}{\Gamma \vdash \lambda x.\, e : A \to B \hookrightarrow \lambda x.\, M}\ {\to}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \to B \hookrightarrow M_1 \qquad \Gamma \vdash e_2 : A \hookrightarrow M_2}{\Gamma \vdash e_1\, e_2 : B \hookrightarrow M_1\, M_2}\ {\to}\mathrm{E}$$

$$\frac{\Gamma \vdash e : A_1 \hookrightarrow M_1 \qquad \Gamma \vdash e : A_2 \hookrightarrow M_2}{\Gamma \vdash e : A_1 \wedge A_2 \hookrightarrow (M_1\,,\,M_2)}\ \wedge\mathrm{I} \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2 \hookrightarrow M}{\Gamma \vdash e : A_k \hookrightarrow \mathbf{proj}_k\, M}\ \wedge\mathrm{E}_k$$

$$\frac{\Gamma \vdash e : A_k \hookrightarrow M}{\Gamma \vdash e : A_1 \vee A_2 \hookrightarrow \mathbf{inj}_k\, M}\ \vee\mathrm{I}_k$$

$$\frac{\Gamma \vdash e_0 : A \hookrightarrow M_0 \qquad \Gamma, x : A \vdash \mathcal{E}[x] : C \hookrightarrow N}{\Gamma \vdash \mathcal{E}[e_0] : C \hookrightarrow (\lambda x.\, N) M_0}\ \mathsf{direct}$$

$$\frac{\Gamma \vdash e_0 : A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{l}\Gamma, x_1 : A_1 \vdash \mathcal{E}[x_1] : C \hookrightarrow N_1 \\ \Gamma, x_2 : A_2 \vdash \mathcal{E}[x_2] : C \hookrightarrow N_2\end{array}}{\Gamma \vdash \mathcal{E}[e_0]\ :\ C \ \hookrightarrow\ \mathbf{case}\ M_0\ \mathbf{of\ inj}_1\, x_1 \Rightarrow N_1 \mathbin{|} \mathbf{inj}_2\, x_2 \Rightarrow N_2}\ \vee\mathrm{E}$$

Fig. 8. Elaboration typing rules.

$$\begin{array}{rcl}
|\top| & = & \mathsf{unit} \\
|A_1 \to A_2| & = & |A_1| \to |A_2| \\
|A_1 \wedge A_2| & = & |A_1| * |A_2| \\
|A_1 \vee A_2| & = & |A_1| + |A_2|
\end{array}$$

Fig. 9. Type translation.

More interestingly, given $\Gamma \vdash e : A$ we can always elaborate e, so elaboration is just as expressive as typing:

Theorem 5 (*Completeness of Elaboration*)
If $\Gamma \vdash e : A$ (without using rule sub) then there exists M such that $\Gamma \vdash e : A \hookrightarrow M$.

Proof
By induction on the given derivation; see *elab-complete* in *typeof-elab.elf*. □

Elaboration produces well-typed terms: Any target term M produced by the elaboration rules has the corresponding target type. In the theorem statement, we assume the obvious translation of contexts $|\Gamma|$; for example:

$$\begin{array}{rcl}
|x{:}\top,\ y{:}\top \vee \top| & = & x{:}|\top|,\ y{:}|\top \vee \top| \\
& = & x{:}\mathsf{unit},\ y{:}|\top| + |\top| \\
& = & x{:}\mathsf{unit},\ y{:}\mathsf{unit} + \mathsf{unit}
\end{array}$$

Theorem 6 (*Elaboration Type Soundness*)
If $\Gamma \vdash e : A \hookrightarrow M$ then $|\Gamma| \vdash M : |A|$.

Proof
By induction on the given derivation. For example, the case for direct, which elaborates to an application, applies typeoftm/arrintro and typeoftm/arrelim. Exploiting a bijection between source types and target types, we actually prove $\Gamma \vdash M : A$, interpreting A and types in Γ as target types: $\wedge$ as $*$, etc. See *elab-type-soundness.elf*. □

6.2 Relating source expressions to target terms

Elaboration produces a term that corresponds closely to the source expression: a target term is the same as a source expression, except that the intersection- and union-related aspects of the computation become explicit in the target. For instance, intersection elimination via $\wedge E_2$, implicit in the source program, becomes the explicit projection $\mathbf{proj}_2$. The target term has nearly the same structure as the source; the elaboration rules only insert operations such as $\mathbf{proj}_2$, duplicate subterms such as the e in $\wedge$I, and omit unused parts of merges.

This gives rise to a relatively simple connection between source expressions and target terms—much simpler than a logical relation, which relates all appropriately-typed terms that have the same extensional behavior. In fact, stepping in the target *preserves elaboration typing*, provided we are allowed to step the source expression zero or more times. This consistency result, Theorem 13, needs several lemmas.

Lemma 7
If $e \rightsquigarrow^* e'$ then $\mathcal{E}[e] \rightsquigarrow^* \mathcal{E}[e']$.

Proof
By induction on the number of steps, using a lemma (*step-eval-context*) that $e \rightsquigarrow e'$ implies $\mathcal{E}[e] \rightsquigarrow \mathcal{E}[e']$. See *step*eval-context* in *step-eval-context.elf*. □

Next, we prove inversion properties of unions, intersections, and arrows. Roughly, we want to say that if an expression of union type elaborates to an injection $\mathbf{inj}_k\ M_0$, it also elaborates to M_0. Dually, if an expression of intersection type elaborates to (M_1, M_2), it also elaborates to M_1 and M_2. Similarly, given an expression of arrow type that elaborates to a λ-abstraction, we can step the expression to a λ-abstraction.

Lemma 8 (*Unions/Injections*)
If $\Gamma \vdash e : A_1 \vee A_2 \hookrightarrow \mathbf{inj}_k\ M_0$ then $\Gamma \vdash e : A_k \hookrightarrow M_0$.

Proof
By induction on the given derivation. The only possible cases are merge$_k$ and $\vee I_k$. See *elab-inl* and *elab-inr* in *elab-union.elf*. □

Lemma 9 (*Intersections/Pairs*)
If $\Gamma \vdash e : A_1 \wedge A_2 \hookrightarrow (M_1, M_2)$ then $\Gamma \vdash e : A_1 \hookrightarrow M_1$ and $\Gamma \vdash e : A_2 \hookrightarrow M_2$.

Proof
By induction on the given derivation; the only possible cases are $\wedge$I and merge_k. See *elab-sect.elf*. $\square$

Lemma 10 (*Arrows/Lambdas*)
If $\cdot \vdash e : A \to B \hookrightarrow \lambda x.\, M_0$ then there exists e_0 such that $e \rightsquigarrow^* \lambda x.\, e_0$ and $x : A \vdash e_0 : B \hookrightarrow M_0$.

Proof
By induction on the given derivation; the only possible cases are $\to$I and merge_k. We show the merge_1 case:

- **Case** merge_1:
$$\mathcal{D} :: \dfrac{\cdot \vdash e_1 : A \to B \hookrightarrow \lambda x.\, M_0}{\cdot \vdash e_1 ,, e_2 : A \to B \hookrightarrow \lambda x.\, M_0}$$
By i.h., there exists e_0 such that $e_1 \rightsquigarrow^* \lambda x.\, e_0$ and $x : A \vdash e_0 : B \hookrightarrow M_0$.
By rule 'step/merge1', $(e_1 ,, e_2) \rightsquigarrow e_1$.
Therefore $(e_1 ,, e_2) \rightsquigarrow^* \lambda x.\, e_0$, which was to be shown.

See *elab-arr.elf*. $\square$

Our last interesting lemma shows that if an expression e elaborates to a target value W, we can step e to some value v that also elaborates to W.

Lemma 11 (*Value monotonicity*)
If $\Gamma \vdash e : A \hookrightarrow W$ then there exists v such that $e \rightsquigarrow^* v$ where $\Gamma \vdash v : A \hookrightarrow W$.

Proof
By induction on the given derivation. The most interesting case is for $\wedge$I.

- **Case** $\wedge$I:
$$\mathcal{D} :: \dfrac{\cdot \vdash e : A_1 \hookrightarrow W_1 \qquad \cdot \vdash e : A_2 \hookrightarrow W_2}{\cdot \vdash e : A_1 \wedge A_2 \hookrightarrow (W_1 , W_2)}$$
Applying the induction hypothesis to each premise yields v_1 and v_2 such that $e \rightsquigarrow^* v_1$ and $e \rightsquigarrow^* v_2$.
Now we need to find a value v such that $\cdot \vdash v : A_1 \wedge A_2 \hookrightarrow (W_1 , W_2)$. So far we only have v_1 and v_2, which may be distinct; but we need a single value v. But we can apply rule 'step/split': $e \rightsquigarrow (e ,, e)$. Repeatedly applying 'step/merge1' gives $(e ,, e) \rightsquigarrow^* (v_1 ,, e)$; likewise, 'step/merge2' gives $(v_1 ,, e) \rightsquigarrow^* (v_1 ,, v_2)$:
$$e \rightsquigarrow (e ,, e) \rightsquigarrow^* (v_1 ,, e) \rightsquigarrow^* (v_1 ,, v_2)$$
Therefore $e \rightsquigarrow^* (v_1 ,, v_2)$. Let $v = (v_1 ,, v_2)$.
By merge_1, $\cdot \vdash v_1 ,, v_2 : A_1 \hookrightarrow W_1$. By merge_2, $\cdot \vdash v_1 ,, v_2 : A_2 \hookrightarrow W_2$.
Then $\wedge$I gives $\cdot \vdash v_1 ,, v_2 : A_1 \wedge A_2 \hookrightarrow (W_1 , W_2)$.

In the merge_k case on a merge $e_1 ,, e_2$, we apply the induction hypothesis to e_k, giving $e_k \rightsquigarrow^* v$. By rule 'step/unmerge', $e_1 ,, e_2 \rightsquigarrow e_k$, from which $e_1 ,, e_2 \rightsquigarrow^* v$.
See *value-mono.elf*. $\square$

Lemma 12 (*Substitution*)
If $\Gamma, x : A \vdash e : B \hookrightarrow M$ and $\Gamma \vdash v : A \hookrightarrow W$ then $\Gamma \vdash [v/x]e : B \hookrightarrow [W/x]M$.

Proof
By induction on the first derivation. Twelf's higher-order abstract syntax gives us this substitution lemma for free. □

6.3 Consistency

The consistency theorem below is the linchpin: given e that elaborates to M, we can preserve the elaboration relationship even after stepping M, though we may have to step e some number of times as well. The expression e and term M, in general, step at different speeds:

- M steps while e doesn't—for example, if M is $\textbf{proj}_1\ (W_1, W_2)$ and steps to W_1, there is nothing to do in e because the projection corresponds to the *implicit* elimination in rule $\wedge E_1$;
- e may step *more* than M—for example, if e is $(v_1 ,, v_2)\, v$ and M is $(\lambda x. x)\, W$, then M β-reduces to W, but e must first 'step/unmerge' to the appropriate v_k, yielding $v_k\, v$, and *then* apply 'step/beta'.

(Note that the converse—if $e \leadsto e'$ then $M \mapsto^* M'$—does not hold: we could pick the wrong half of a merge and get a source expression with no particular relation to M.)

Theorem 13 (*Consistency*)
If $\cdot \vdash e : A \hookrightarrow M$ and $M \mapsto M'$
then there exists e' such that $e \leadsto^* e'$ and $\cdot \vdash e' : A \hookrightarrow M'$.

Proof
By induction on the derivation $\mathcal{D}$ of $\cdot \vdash e : A \hookrightarrow M$. We show several cases here; the full proof is in `consistency.elf`.

- **Case** var, $\top$I, $\rightarrow$I: Impossible because M cannot step.

- **Case** $\wedge$I:
$$\mathcal{D} :: \frac{\cdot \vdash e : A_1 \hookrightarrow M_1 \qquad \cdot \vdash e : A_2 \hookrightarrow M_2}{\cdot \vdash e : A_1 \wedge A_2 \hookrightarrow (M_1, M_2)}$$
By inversion, either $M_1 \mapsto M_1'$ or $M_2 \mapsto M_2'$. Suppose the former (the latter is similar). By i.h., $e \leadsto^* e_1'$ and $\cdot \vdash e_1' : A_1 \hookrightarrow M_1'$. By 'step/split', $e \leadsto e ,, e$. Repeatedly applying 'step/merge1' gives $e ,, e \leadsto^* e_1' ,, e$.
For typing, apply merge$_1$ with premise $\cdot \vdash e_1' : A_1 \hookrightarrow M_1'$ and merge$_2$ with premise $\cdot \vdash e : A_2 \hookrightarrow M_2$.
Finally, by $\wedge$I, we have $\cdot \vdash e_1' ,, e : A_1 \wedge A_2 \hookrightarrow (M_1', M_2)$.

- **Case** $\wedge E_k$:
$$\mathcal{D} :: \frac{\cdot \vdash e : A_1 \wedge A_2 \hookrightarrow M_0}{\cdot \vdash e : A_k \hookrightarrow \textbf{proj}_k\, M_0}$$
If $\textbf{proj}_k\, M_0 \mapsto \textbf{proj}_k\, M_0'$ with $M_0 \mapsto M_0'$, use the i.h. and apply $\wedge E_k$.

If $M_0 = (W_1, W_2)$ and $\mathbf{proj}_k\ M_0 \mapsto W_k$, use Lemma 9, yielding $\Gamma \vdash e : A_k \hookrightarrow W_k$.

- **Case** merge_k:
$$\mathcal{D} :: \frac{\cdot \vdash e_k : A \hookrightarrow M}{\cdot \vdash e_1 ,, e_2 : A \hookrightarrow M}$$
By i.h., $e_k \rightsquigarrow^* e'$ and $\cdot \vdash e' : A \hookrightarrow M'$. By rule 'step/unmerge', $e_1 ,, e_2 \rightsquigarrow e_k$. Therefore $e_1 ,, e_2 \rightsquigarrow^* e'$.

- **Case** $\to$E:
$$\mathcal{D} :: \frac{\cdot \vdash e_1 : A \to B \hookrightarrow M_1 \qquad \cdot \vdash e_2 : A \hookrightarrow M_2}{\cdot \vdash e_1\ e_2 : B \hookrightarrow M_1\ M_2}$$
We show one of the harder subcases (`consistency/app/beta` in `consistency.elf`). In this subcase, $M_1 = \lambda x.\, M_0$ and M_2 is a value, with $M_1\ M_2 \mapsto [M_2/x]M_0$. We use several easy lemmas about stepping; for example, `step*app1` says that if $e_1 \rightsquigarrow^* e_1'$ then $e_1\ e_2 \rightsquigarrow^* e_1'\ e_2$.

Elab1 ::	$\cdot \vdash e_1 : A \to B \hookrightarrow \lambda x.\, M_0$	Subd.
ElabBody ::	$x : A \vdash e_0 : B \hookrightarrow M_0$	By Lemma 10
StepsFun ::	$e_1 \rightsquigarrow^* \lambda x.\, e_0$	"
StepsApp ::	$e_1\ e_2 \rightsquigarrow^* (\lambda x.\, e_0) e_2$	By `step*app1`
Elab2 ::	$\cdot \vdash e_2 : A \hookrightarrow M_2$	Subd.
	M_2 value	Above
Elab2′ ::	$\cdot \vdash e_2 \rightsquigarrow^* v_2$	By Lemma 11
	$\cdot \vdash v_2 : A \hookrightarrow M_2$	"
	$(\lambda x.\, e_0) e_2 \rightsquigarrow^* (\lambda x.\, e_0) v_2$	By `step*app2`
	$e_1\ e_2 \rightsquigarrow^* (\lambda x.\, e_0) v_2$	By `step*append`
	$(\lambda x.\, e_0) v_2 \rightsquigarrow [v_2/x]e_0$	By 'step/beta'
StepsAppBeta ::	$e_1\ e_2 \rightsquigarrow^* [v_2/x]e_0$	By `step*snoc`
ElabBody ::	$x : A \vdash e_0 : B \hookrightarrow M_0$	Above
	$\cdot \vdash [v_2/x]e_0 : B \hookrightarrow [M_2/x]M_0$	By Lemma 12 (Elab2′)

Theorem 14 (*Multistep Consistency*)
If $\cdot \vdash e : A \hookrightarrow M$ and $M \mapsto^* W$ then there exists v such that $e \rightsquigarrow^* v$ and $\cdot \vdash v : A \hookrightarrow W$.

Proof
By induction on the derivation of $M \mapsto^* W$.

If M is some value W then, by Lemma 11, e is some value v. The source expression e steps to itself in zero steps, so $v \rightsquigarrow^* v$, and $\cdot \vdash v : A \hookrightarrow W$ is given ($e = v$ and $M = W$).

Otherwise, we have $M \mapsto M'$ where $M' \mapsto^* W$. We want to show $\cdot \vdash e' : A \hookrightarrow M'$, where $e \rightsquigarrow^* e'$. By Theorem 13, either $\cdot \vdash e : A \hookrightarrow M'$, or $e \rightsquigarrow e'$ and $\cdot \vdash e' : A \hookrightarrow M'$.

- If $\cdot \vdash e : A \hookrightarrow M'$, let $e' = e$, so $\cdot \vdash e' : A \hookrightarrow M'$ and $e \rightsquigarrow^* e'$ in zero steps.
- If $e \rightsquigarrow e'$ and $\cdot \vdash e' : A \hookrightarrow M'$, we can use the i.h., showing that $e' \rightsquigarrow^* v$ and $\cdot \vdash v : A \hookrightarrow W$.

See `consistency*` in `consistency.elf`. □

6.4 Summing up

Theorem 15 (*Static Semantics*)
If $\cdot \vdash e : A$ (using any of the rules in Figure 4) then there exists e' such that $\cdot \vdash e' : A \hookrightarrow M$ and $\cdot \vdash M : |A|$.

Proof
By Theorems 1 (coercion), 5 (completeness of elaboration), and 6 (elaboration type soundness). □

Theorem 16 (*Dynamic Semantics*)
If $\cdot \vdash e : A \hookrightarrow M$ and $M \mapsto^* W$ then there is a source value v such that $e \rightsquigarrow^* v$ and $\cdot \vdash v : A$.

Proof
By Theorems 14 (multistep consistency) and 4. □

Recalling the diagram in Figure 1, Theorem 16 shows that it commutes.

Both theorems are stated and proved in `summary.elf`. Combined with a run of the target program, $M \mapsto^* W$, they show that elaborated programs are consistent with source programs.

6.5 The value restriction

Let's turn for a moment to parametric polymorphism. The natural rule for introducing a polymorphic type (sometimes distinguished as a *type scheme*) would be

$$\frac{\Delta, \alpha\ \mathsf{type} \vdash e : A}{\Delta \vdash e : \forall \alpha.\, A}\ \forall\mathrm{I}$$

However, in a call-by-value semantics with mutable references, this rule is unsound, as shown by this example:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{let}\ r = (\textbf{ref}\ \mathsf{Nil}) : \forall \alpha.\, \textbf{ref}\ (\textbf{list}\ \alpha)\ \textbf{in}} \\
\quad r := [3]; & \text{—by instantiating } \alpha \text{ with int} \\
\quad (!r) : \textbf{list}\ \mathsf{bool} & \text{—by instantiating } \alpha \text{ with bool}
\end{array}$$

Here, `!r` will evaluate to `[3]`, which is a list of integers, not a list of booleans. The original specification of Standard ML was unsound, since it permitted examples along these lines. Various solutions were proposed; the revised Definition (Milner *et al.*, 1997,

p. 86) followed Wright (1995), who proposed restricting ∀-introduction to values v:

$$\frac{\Delta, \alpha\ \mathsf{type} \vdash v : A}{\Delta \vdash v : \forall\alpha.\, A}\ \forall\mathrm{I}\ \textit{(≈Wright)}$$

A few years later, Davies and Pfenning (2000) showed that the then-standard rule for intersection introduction (that is, our ∧I) was unsound in a call-by-value semantics in the presence of effects—specifically, mutable references. Here is an example, essentially the same as theirs. Assume a base type nat with values $0, 1, 2, \ldots$ and a type pos of strictly positive naturals with values $1, 2, \ldots$; assume $\mathsf{pos} \leqslant \mathsf{nat}$.

$$\begin{array}{l} \mathbf{let}\, r = (\mathbf{ref}\ 1) : (\mathbf{ref}\ \mathsf{nat}) \wedge (\mathbf{ref}\ \mathsf{pos})\ \mathbf{in} \\ \quad r := 0; \\ \quad (!r) : \mathsf{pos} \end{array}$$

Using the unrestricted ∧I rule, r has type $(\mathbf{ref}\ \mathsf{nat}) \wedge (\mathbf{ref}\ \mathsf{pos})$; using $\wedge\mathrm{E}_1$ yields $r : \mathbf{ref}\ \mathsf{nat}$, so the write $r := 0$ is well-typed; using $\wedge\mathrm{E}_2$ yields $r : \mathbf{ref}\ \mathsf{pos}$, so the read $!r$ produces a pos. In an unelaborated setting, this typing is unsound: $(\mathbf{ref}\ 1)$ creates a single cell containing 1, which is overwritten with 0; then $!r \rightsquigarrow 0$, which does not have type pos.

Noting the apparent similarity of this problem with ∧-introduction to the earlier problem with ∀-introduction, Davies and Pfenning proposed an analogous value restriction: an ∧-introduction rule that only types values v. This rule is sound with mutable references:

$$\frac{v : A_1 \qquad v : A_2}{v : A_1 \wedge A_2}\ \wedge\mathrm{I}\ \textit{(Davies and Pfenning)}$$

In our elaboration system, however, the problematic example above is sound, because our ∧I elaborates $\mathbf{ref}\ 1$ to two distinct expressions, which create two unaliased cells:

$$\frac{\mathbf{ref}\ 1 : \mathbf{ref}\ \mathsf{nat} \hookrightarrow \mathbf{ref}\ 1 \qquad \mathbf{ref}\ 1 : \mathbf{ref}\ \mathsf{pos} \hookrightarrow \mathbf{ref}\ 1}{\mathbf{ref}\ 1 : \mathbf{ref}\ \mathsf{nat} \wedge \mathbf{ref}\ \mathsf{pos} \hookrightarrow (\mathbf{ref}\ 1, \mathbf{ref}\ 1)}\ \wedge\mathrm{I}$$

Thus, the example elaborates to

$$\begin{array}{l} \mathbf{let}\, r = (\mathbf{ref}\ 1, \mathbf{ref}\ 1)\ \mathbf{in} \\ \quad (\mathbf{proj}_1\ r) := 0; \\ \quad (!\,\mathbf{proj}_2\ r) : \mathsf{pos} \end{array}$$

which is well-typed, but does not "go wrong" in the type-safety sense: the assignment writes to the first cell ($\wedge\mathrm{E}_1$), and the dereference reads the second cell ($\wedge\mathrm{E}_2$), which still contains the original value 1. The restriction-free ∧I thus appears sound in our setting. Being *sound* is not the same as being *useful*, though; such behavior is less than intuitive, as we discuss in the next section.

```
val mul = Int.*
val toString = Int.toString

val mul = mul ,, Real.*   (* shadows earlier 'mul' *)
val toString = toString ,, Real.toString

val square : (int → int) ∧ (real → real)
val square = fn x ⇒ x * x

val _ = print (toString (mul (0.5, 300.0)) ^ "; ")
val _ = print (toString (square 9) ^ "; ")
val _ = print (toString (square 0.5) ^ "\n")
```

Output of target program after elaboration: `150.0; 81; 0.25`

Fig. 10. Example of overloading.

7 Coherence

The merge construct, while simple and powerful, has serious usability issues when the parts of the merge have overlapping types. Or, more accurately, when their types would overlap—have nonempty intersection—in a merge-free system; in our system, *all* intersections $A \wedge B$ of nonempty A, B are nonempty: if $v_A : A$ and $v_B : B$ then $v_A ,, v_B : A \wedge B$ by merge_1, merge_2, and $\wedge$I.

According to the elaboration rules, the expression $0 ,, 1$ (checked against nat) could elaborate to either 0 or 1. Our implementation would elaborate $0 ,, 1$ to 0, because it tries the left part 0 first. Arguably, this is better behavior than actual randomness, but hardly helpful to the programmer. Perhaps even more confusingly, suppose we check $0 ,, 1$ against $\mathsf{pos} \wedge \mathsf{nat}$, where pos and nat are as in Section 6.5. Our implementation elaborates $0 ,, 1$ to $(1, 0)$, but elaborates $1 ,, 0$ to $(1, 1)$.

Since the behaviour of the target program depends on the particular elaboration typing used, the system lacks *coherence* (Reynolds, 1991). To recover a coherent semantics, we could limit merges according to their surface syntax, as Reynolds did in Forsythe, but crafting an appropriate syntactic restriction depends on details of the type system, which is not robust as the type system is extended. A more general approach would be to reject (or warn about) merges in which more than one part checks against the same type, or the same part of an intersection type; we will return to this in Section 11.

Leaving merges aside, the mere fact that $\wedge$I elaborates the expression twice creates problems with mutable references, as we saw in Section 6.5. To address this, we could revive the value restriction in $\wedge$I, at least for expressions whose types might overlap.

8 Applying intersections and unions

8.1 Overloading

A very simple use of unrestricted intersections is to "overload" operations such as multiplication and conversion of data to printable form. SML provides overloading only for a (syntactically) fixed set of built-in operations; it is not possible to write an overloaded `square` function, such as ours in Figure 10.

Unlike Standard ML, we provide a convenient syntax for type annotations that conforms to SML module signatures. For example, **val** `square` : ... is a type annotation that applies to the subsequent declaration of `square`. (Previous versions of our system, including the one described in Dunfield (2012), used a different syntax that allowed source programs to be valid Standard ML programs—a futile goal in the context of unrestricted intersection and union types.)

In its present form, this idiom is less powerful than type classes (Wadler and Blott, 1989). We could extend `toString` for lists, which would handle lists of integers and lists of reals, but not lists of lists; the version of `toString` for lists would use the *earlier* occurrence of `toString`, defined for integers and reals only. Adding a mechanism for naming a type and then "unioning" it, recursively, is future work.

8.2 Records

Reynolds (1996) developed an encoding of records using intersection types and his version of the merge construct; similar ideas appear in Castagna *et al.* (1995). Though straightforward, this encoding is more expressive than SML records.

The idea is to add single-field records as a primitive notion, through a type $\{\texttt{fld} : A\}$ with introduction form $\{\texttt{fld}{=}\,e\}$ and the usual eliminations (explicit projection and pattern matching). Once this is done, the multifield record type $\{\texttt{fld1} : A_1\,,\ \texttt{fld2} : A_2\}$ is simply $\{\texttt{fld1} : A_1\} \wedge \{\texttt{fld2} : A_2\}$, and it can be introduced by a merge:

$$\{\texttt{fld1}{=}\,e_1\},, \{\texttt{fld2}{=}\,e_2\}$$

More standard concrete syntax, such as $\{\texttt{fld1}{=}\,e_1\,,\ \texttt{fld2}{=}\,e_2\}$, can be handled trivially during parsing.

With subtyping on intersections, we get the desired behavior of what SML calls "flex records"—records with some fields not listed—with fewer of SML's limitations. Using this encoding, a function that expects a record with fields `x` and `y` can be given *any* record that has at least those fields, whereas SML only allows one fixed set of fields. For example, the code in Figure 11 is legal in our language but not in SML.

One problem with this approach is that expressions with duplicated field names are accepted. This is part of the larger issue discussed in Section 7.

8.3 Heterogeneous data

A common argument for dynamic typing over static typing is that heterogeneous data structures are more convenient. For example, dynamic typing makes it very easy to create and manipulate lists containing both integers and strings. The penalty is the loss of compile-time invariant checking. Perhaps the lists should contain integers and strings, but not booleans; such an invariant is not expressible in traditional dynamic typing.

A common rebuttal from advocates of static typing is that it is easy to simulate dynamic typing in static typing. Want a list of integers and strings? Just declare a datatype

```
datatype int_or_string = Int of int
                       | String of string
```

```
val get_xy : {x:int, y:int} → int * int
fun get_xy r =
  (#x(r), #y(r))

val tupleToString : int * int → string
fun tupleToString (x, y) =
 "(" ^ Int.toString x ^ "," ^ Int.toString y ^ ")"

val rec1 = {y = 11, x = 1}
val rec2 = {x = 2, y = 22, extra = 100}
val rec3 = {x = 3, y = 33, other = "a string"}

val _ = print("get_xy rec1 = " ^ tupleToString (get_xy rec1) ^ "\n")
val _ = print("get_xy rec2 = " ^ tupleToString (get_xy rec2)
          ^ " (extra = " ^ Int.toString #extra(rec2) ^ ")\n")
val _ = print("get_xy rec3 = " ^ tupleToString (get_xy rec3)
          ^ " (other = " ^ #other(rec3) ^ ")\n")
```

Output of target program after elaboration:

```
get_xy rec1 = (1,11)
get_xy rec2 = (2,22) (extra = 100)
get_xy rec3 = (3,33) (other = a string)
```

Fig. 11. Example of flexible multifield records.

and use lists of type list int_or_string[3]. This guarantees the invariant that the list has only integers and strings, but is unwieldy: each new element must be wrapped in a constructor, and operations on the list elements must unwrap the constructor, even when those operations accept both integers and strings (such as a function of type $(\mathsf{int} \rightarrow \mathsf{string}) \wedge (\mathsf{string} \rightarrow \mathsf{string})$).

In this situation, our approach provides the compile-time invariant checking of static typing *and* the transparency of dynamic typing. The type of list elements (if we bother to declare it) is just a union type:

type int_union_string = int $\vee$ string

The elaboration process transforms programs that use `int_union_string` into programs that use `int_or_string`.

Along these lines, we use in Figure 12 a type dyn, defined as int $\vee$ real $\vee$ string. It would be useful to also allow lists, but the current implementation lacks recursive types of a form that could express "dyn = ... $\vee$ list dyn".

Note that the η-expansion in `toString` is necessary: if we instead wrote

val toString = Int.toString ,, (**fn** s ⇒ s : string) ,, Real.toString

we would attempt to check the merge against the type (int $\vee$ real $\vee$ string)$\rightarrow$string. However, we cannot apply $\rightarrow$E because the term is a merge, not a λ-abstraction. We also cannot apply merge because no single part of the merge can handle int $\vee$ real $\vee$ string—each part handles only one type from the union. In the η-expanded version, the variable x has type int $\vee$ real $\vee$ string and appears in an evaluation position (recall that a merge of values is a value), so we can apply $\vee$E.

[3] In our syntax, type constructors are given first, as in Haskell.

```
datatype list 'a
datacon nil : -all 'a- list 'a
datacon :: : -all 'a- 'a * list 'a → list 'a

type dyn = int ∨ real ∨ string

val toString : dyn → string
fun toString x =
  (Int.toString ,,
   (fn s ⇒ s : string) ,,
   Real.toString) x

val hetListToString : list dyn → string
fun hetListToString xs = case xs of
    nil  ⇒ "nil"
  | h::t ⇒ (toString h) ^ "::"
        ^  (hetListToString t)

val _ = print "\n\n"
val _ = print (hetListToString [1, 2, "what", 3.14159, 4, "why"])
val _ = print "\n\n\n"
```

Output of target program after elaboration: `1::2::what::3.14159::4::why::nil`

Fig. 12. Example of heterogeneous data (*dyn.sdml*).

Alternatively, we could try to check the unexpanded version against an intersection of arrows:

```
val toString : (int → string) ∧ (real → string) ∧ (string → string)
val toString = Int.toString ,, (fn s ⇒ s : string) ,, Real.toString
```

This typechecks, but is less than ideal: while the subtyping relation

$$(\mathsf{int} \to \mathsf{string}) \wedge (\mathsf{real} \to \mathsf{string}) \wedge (\mathsf{string} \to \mathsf{string}) \leqslant (\mathsf{int} \vee \mathsf{real} \vee \mathsf{string}) \to \mathsf{string}$$

is sound, it is not derivable in our system of subtyping rules, so we cannot pass this version of toString to a function expecting an argument of type $(\mathsf{int} \vee \mathsf{real} \vee \mathsf{string}) \to \mathsf{string}$. It does, however, suffice to make the rest of the example typecheck: In the body of hetListToString we have the application (toString h), where h has union type, so we can apply $\vee$E. Even if we extended the subtyping rules, the user would still have to write out the intersection, instead of simply writing dyn → string.

9 Implementation

Our prototype implementation, called Stardust, is faithful to the spirit of the elaboration rules above, but is substantially richer. It builds on an earlier implementation (Dunfield, 2007) of a typechecker for a subset of core Standard ML with support for inductive datatypes, products, intersections, unions, refinement types, and indexed types, extended with support for (first-class) polymorphism (Dunfield, 2009). The current implementation does not fully support some of these features; support for first-class polymorphism looks hardest, since Standard ML compilers cannot even compile programs that use higher-rank predicative polymorphism. Elaborating programs that use ML-style prenex polymorphism should work, but we currently lack any significant testing, much less proof, to back that up.

Our implementation does currently support merges, intersections, and unions, a top type, a bottom (empty) type, single-field records and encoded multifield records (Section 8.2), and inductive datatypes. It also supports a form of exception; the expression **raise** e does not return a value, and checks against the bottom type. Support for refinement and indexed types is spotty, but some of the examples that worked in the old system (Dunfield, 2007) work in the new one. Stardust includes both refinement and unrestricted versions of intersection and union; however, mixing them in the same program is not supported (and appears nontrivial to solve; see the discussion in Section 11).

The implementation, including examples, can be downloaded from `stardust.qc.com`.

9.1 Bidirectional typechecking

Our implementation uses *bidirectional typechecking* (Pierce and Turner, 2000; Dunfield and Pfenning, 2004; Dunfield and Krishnaswami, 2013). This technique offers two major benefits over Damas–Milner type inference: it works for many type systems where annotation-free inference is undecidable, and it seems to produce better-localized error messages. See Dunfield and Krishnaswami (2013) for references.

Bidirectional typechecking does need more type annotations than Damas–Milner inference. However, by following the approach of Dunfield and Pfenning (2004), annotations are never needed except on redexes. The implemented typechecker allows some annotations on redexes to be omitted, as well.

The basic idea of bidirectional typechecking is to separate the activity of checking an expression against a known type from the activity of synthesizing a type from the expression itself:

$$\begin{array}{ll} \Gamma \vdash e \Leftarrow A & e \text{ checks against known type } A \\ \Gamma \vdash e \Rightarrow A & e \text{ synthesizes type } A \end{array}$$

In the checking judgment, Γ, e and A are inputs to the typing algorithm, which either succeeds or fails. In the synthesis judgment, Γ and e are inputs and A is output (assuming synthesis does not fail). The direction of the arrows ($\Leftarrow$, $\Rightarrow$) corresponds to the flow of type information.

Syntactically speaking, crafting a bidirectional type system from a type assignment system (like the one in Figure 4) is largely a matter of taking the colons in the $\Gamma \vdash e : A$ judgments and replacing some with "$\Leftarrow$" and some with "$\Rightarrow$". Except for the rules for merges, all our typing rules can be found in Dunfield and Pfenning (2004), which argued that introduction rules should check and elimination rules should synthesize. For functions, this leads to the rules

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

The merge rule, however, neither introduces nor eliminates. We implement the obvious checking rule (which, in practice, always tries to check against e_1 and, if that fails, against e_2):

$$\frac{\Gamma \vdash e_k \Leftarrow A}{\Gamma \vdash e_1 ,, e_2 \Leftarrow A}\ \mathsf{merge}_k{\Leftarrow}$$

$\boxed{\Gamma \vdash e \Leftarrow A}$ Source expression e checks against source type A

$\boxed{\Gamma \vdash e \Rightarrow A}$ Source expression e synthesizes source type A

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x \Rightarrow A}\ \mathsf{var}{\Rightarrow} \qquad \frac{\Gamma \vdash e_k \Leftarrow A}{\Gamma \vdash e_1 ,, e_2 \Leftarrow A}\ \mathsf{merge}_k{\Leftarrow}$$

$$\frac{\Gamma \vdash e_k \Rightarrow A}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A}\ \mathsf{merge}_k{\Rightarrow} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A_1 \qquad \Gamma \vdash e_2 \Rightarrow A_2}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A_1 \wedge A_2}\ \mathsf{merge}_{\wedge}{\Rightarrow}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow A}{\Gamma \vdash \mathbf{fix}\, x.\, e \Leftarrow A}\ \mathsf{fix}{\Leftarrow} \qquad \frac{}{\Gamma \vdash v \Leftarrow \top}\ \top\mathrm{I}{\Leftarrow}$$

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

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

$$\frac{\Gamma \vdash e_0 \Rightarrow A \qquad \Gamma, x : A \vdash \mathcal{E}[x] \Leftarrow C}{\Gamma \vdash \mathcal{E}[e_0] \Leftarrow C}\ \mathsf{direct}$$

$$\frac{\Gamma \vdash e \Leftarrow A_k}{\Gamma \vdash e \Leftarrow A_1 \vee A_2}\ \vee\mathrm{I}_k{\Leftarrow} \qquad \frac{\Gamma \vdash e_0 \Rightarrow A_1 \vee A_2 \qquad \begin{array}{c}\Gamma, x_1 : A_1 \vdash \mathcal{E}[x_1] \Leftarrow C \\ \Gamma, x_2 : A_2 \vdash \mathcal{E}[x_2] \Leftarrow C\end{array}}{\Gamma \vdash \mathcal{E}[e_0] \Leftarrow C}\ \vee\mathrm{E}$$

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A \le B ::: e_{\mathrm{coerce}}}{\Gamma \vdash e \Leftarrow B}\ \mathsf{sub} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}\ \mathsf{anno}$$

Fig. 13. Bidirectional typing for the source language.

Since it can be inconvenient to annotate merges, we also implement synthesis rules, including one that can synthesize an intersection ($\mathsf{merge}_{\wedge}{\Rightarrow}$); see Figure 13.

Given a bidirectional typing derivation, it is generally easy to show that a corresponding type assignment exists: replace all "$\Rightarrow$" and "$\Leftarrow$" with ":" (and erase explicit type annotations from the expression). In the other direction, given a type assignment derivation, we can show that a bidirectional derivation exists after *adding* some type annotations. Bidirectional typing is certainly incomplete in the sense that it cannot synthesize a type for every (well-typed) unannotated term—but since type assignment for most, if not all, intersection type systems is undecidable (Coppo *et al.*, 1981), completeness is not achievable.

That said, the system in Figure 13 seems excessively incomplete; for example, no rule can synthesize a type for $()$, nor for a function $\lambda x.\, e$, even in cases where the body e synthesizes and does not use x (or makes it obvious what type x must have). More elaborate systems of bidirectional typechecking require fewer annotations, and can support parametric polymorphism; the implementation is based on an algorithm given by Dunfield (2009), but a better reference is the simpler approach developed by Dunfield and Krishnaswami (2013).

The implementation also transforms programs to a variant of let-normal form before checking them, which (partly) addresses one source of backtracking during typechecking: the choice of evaluation context in the $\vee$E rule. This transformation is described in Dunfield (2011), with the caveat that the system considered there lacks the merge rules.

9.2 Performance

Intersection typechecking is PSPACE-hard (Reynolds, 1996). In practice, we elaborate the examples in Figures 10–12 in less than a second, but they are very small. On somewhat larger examples, such as those discussed by Dunfield (2007), the non-elaborating version of Stardust could take minutes, thanks to heavy use of backtracking search (trying $\wedge E_1$ then $\wedge E_2$, etc.) and the need to check the same expression against different types ($\wedge$I) or with different assumptions ($\vee$E). Elaboration doesn't help with this, but it shouldn't hurt by more than a constant factor: the shapes of the derivations and the labor of backtracking remain the same.

To scale up the approach to larger programs, we will need to consider how to efficiently represent elaborated intersections and unions. Like the theoretical development, the implementation has two-way intersection and union types, so the type $A_1 \wedge A_2 \wedge A_3$ is parsed as $(A_1 \wedge A_2) \wedge A_3$, which becomes $(A_1 * A_2) * A_3$. A flattened representation $A_1 * A_2 * A_3$ would be more efficient, except when the program uses values of type $(A_1 \wedge A_2) \wedge A_3$ where values of type $A_1 \wedge A_2$ are expected; in that case, nesting the product allows the inner pair to be passed directly with no reboxing. Symmetry is also likely to be an issue: passing $v : A_1 \wedge A_2$ where $v : A_2 \wedge A_1$ is expected requires building a new pair. Here, it may be helpful to put the components of intersections into a canonical order.

The foregoing applies to unions as well—introducing a value of a three-way union can lead to two injections, and so on.

10 Related work

Intersections were originally developed by Coppo *et al.* (1981) and Pottinger (1980), among others; Hindley (1992) gives a useful introduction and bibliography. Building on Pottinger's work, Lopez-Escobar (1985) called intersection a *proof-functional connective* (as opposed to truth-functional) and defined a variant of the sequent calculus with intersection instead of conjunction. In that system, intersection introduction is allowed only when the two subderivations have a similar structure, roughly analogous to the requirement that each subderivation of our intersection has the same subject term. Work on union types began later (MacQueen *et al.*, 1986); a key paper on type assignment for unions is Barbanera *et al.* (1995).

Forsythe. In the late 1980s, Reynolds invented Forsythe (Reynolds, 1996), the first practical programming language based on intersection types. (The citation year 1996 is the date of the revised description of Forsythe; the core ideas are found in Reynolds 1988.) In addition to an unmarked introduction rule like $\wedge$I, the Forsythe type system includes rules for typing a construct p_1 , p_2—"a construction for intersecting or 'merging' meanings" (Reynolds, 1996, p. 24). Roughly analogous to $e_1 ,, e_2$, this construct is used to encode a variety of features, but in Forsythe (unlike our present system) merges can only be used unambiguously. For instance, a record and a function can be merged, but two functions cannot. Forsythe does not have union types.

Pierce's work. Pierce (1991) describes a prototype compiler for a language with intersection and union types that transforms intersections to products, and unions to sums. Pierce's language includes a construct for explicitly eliminating unions. But this construct is only a marker for where to eliminate the union: it has only one branch, so the same term must typecheck under each assumption. Another difference is that this construct is the only way to eliminate a union type in his system, whereas our $\vee$E is marker-free. Intersections, also present in his language, have no explicit introduction construct; the introduction rule is like our $\wedge$I.

The λ&-calculus. Castagna *et al.* (1995) developed the λ&-calculus, which has &-terms—functions whose body is a merge, and whose type is an intersection of arrows. In their semantics, applying a &-term to some argument reduces the term to the branch of the merge with the smallest (compatible) domain. Suppose we have a &-term with two branches, one of type $\mathsf{nat} \rightarrow \mathsf{nat}$ and one of type $\mathsf{pos} \rightarrow \mathsf{pos}$. Applying that &-term to a value of type pos steps to the second branch, because its domain pos is (strictly) a subtype of nat.

Despite the presence of a merge-like construct, their work on the λ&-calculus is markedly different from ours: it gives a semantics to programs directly, and uses type information to do so, whereas we elaborate to a standard term language with no runtime type information. In their work, terms have both *compile-time types* and *run-time types* (the run-time types become more precise as the computation continues); the semantics of applying a &-term depends on the run-time type of the argument to choose the branch. The choice of the *smallest* compatible domain is consistent with notions of inheritance in object-oriented programming, where a class can override the methods of its parent.

Semantic subtyping. Following the λ&-calculus, Frisch *et al.* (2008) investigated a notion of purely semantic subtyping, where the definition of subtyping arises from a model of types, as opposed to the syntactic approach used in our system. They support intersections, unions, function spaces, and even complement. Their language includes a *dynamic type dispatch* which, very roughly, combines a merge with a generalization of our union elimination. Again, the semantics relies on run-time type information.

Flow types. Turbak *et al.* (1997) and Wells *et al.* (2002) use intersections in a system with flow types. They produce programs with *virtual tuples* and *virtual sums*, which correspond to the tuples and sums we produce by elaboration. However, these constructs are internal: nothing in their work corresponds to our explicit intersection and union term constructors, since their system is only intended to capture existing flow properties. They do not compile the virtual constructs into the ordinary ones.

Heterogeneous data and dynamic typing. Several approaches to combining the transparency of dynamic typing and the guarantees of static typing have been investigated. *Soft typing* (Cartwright and Fagan, 1991; Aiken *et al.*, 1994) adds a kind of type inference on top of dynamic typing, but provides no ironclad guarantees. Typed Scheme (Tobin-Hochstadt and Felleisen, 2008), developed to retroactively type Scheme programs, has a flow-sensitive type system with union types, directly supporting heterogeneous data in the style of Section 8.3. Unlike soft typing, Typed Scheme guarantees type safety and provides genuine (even first-class) polymorphism, though programmers are expected to write some annotations.

Type refinements. Restricting intersections and unions to refinements of a single base type simplifies many issues, and is conservative: programs can be checked against refined types, then compiled normally. This approach has been explored for intersections (Freeman and Pfenning, 1991; Davies and Pfenning, 2000), and for intersections and unions (Dunfield and Pfenning, 2003, 2004).

11 Conclusion

We have laid a simple yet powerful foundation for compiling unrestricted intersections and unions: elaboration into a standard functional language. Rather than trying to directly understand the behaviors of source programs, we describe them via their consistency with the target programs.

The most immediate challenge is coherence: While our elaboration approach guarantees type safety of the compiled program, the meaning of the compiled program depends on the particular elaboration typing derivation used; the meaning of the source program is actually implementation-defined.

One possible solution is to restrict typing of merges so that a merge has type A only if *exactly one* branch has type A. We could also partially revive the value restriction, giving nonvalues intersection type only if (to a conservative approximation) both components of the intersection are provably disjoint, in the sense that no merge-free expression has both types. Alternatively, we could introduce a distinct type constructor for "disjoint intersection," which would be well-formed only when its components are provably disjoint. This does not seem straightforward, especially with parametric polymorphism. Consider checking the expression

$$\lambda x.\, \mathbf{let}\, y = (0 ,, x)\, \mathbf{in}\, x$$

against type $\forall\alpha.\, \alpha \to \alpha$. The merge is ambiguous only if α is instantiated with int, so we need to track which types and type variables are (potentially) overlapping. While this seems feasible in special cases, such as polymorphic records—see, for example, Rémy (1989)—it seems highly nontrivial in full generality. But our goal is to use intersections and unions as general mechanisms for encoding language features, so we really should do it in full generality, or not at all.

Another challenge is to reconcile, in spirit and form, the unrestricted view of intersections and unions of this paper with the refinement approach. Elaborating a refinement intersection like $(\mathsf{pos} \to \mathsf{neg}) \wedge (\mathsf{neg} \to \mathsf{pos})$ to a pair of functions seems pointless (unless it can somehow facilitate optimizations in the compiler). The current implementation actually uses different type constructors for "refinement intersection" and unrestricted intersection (and union), which seems to work as long as the two are not mixed. For example, applying a function of type $(\mathsf{pos} \to \mathsf{neg}) \wedge (\mathsf{neg} \to \mathsf{pos})$ to an argument of type $\mathsf{pos} \vee \mathsf{neg}$ is fine: the $\vee$ becomes a sum, and an explicit case analysis picks out the component of the $\wedge$-pair. However, applying such a function to an argument of type $\mathsf{pos} \mathbin{\vee\!\!\!\vee} \mathsf{neg}$—where $\vee\!\!\!\vee$ is refinement union—would require runtime analysis of the argument value to determine whether it had type pos or type neg, since refinement unions are not elaborated to sums. For atomic values like integers, such an analysis seems feasible, but for a refinement like list evenness or bitstring parity, determining the branch of the union would require traversing the entire data structure—a dramatic and non-obvious increase in asymptotic complexity.

Acknowledgments

In 2008, Adam Megacz suggested (after I explained the idea of compiling intersection to product) that one could use an existing ML compiler "as a backend." This version has (I hope) benefited through the comments of the JFP reviewers (as an earlier version did from the ICFP reviewers' suggestions). Finally, I have had useful discussions about this work with Yan Chen, Matthew A. Hammer, Scott Kilpatrick, Neelakantan R. Krishnaswami, and Viktor Vafeiadis.

References

Aiken, A., Wimmers, E. L. & Lakshman, T. K. (1994) Soft typing with conditional types. In *Principles of Programming Languages*, New York: ACM Press, pp. 163–173.

Barbanera, F., Dezani-Ciancaglini, M. & de'Liguoro, U. (1995) Intersection and union types: syntax and semantics. *Inf. Comput.* **119**, 202–230.

Barendregt, H., Coppo, M. & Dezani-Ciancaglini, M. (1983) A filter lambda model and the completeness of type assignment. *J. Symb. Log.* **48**(4), 931–940.

Cartwright, R. & Fagan, M. (1991) Soft typing. In *Programming Language Design and Implementation*, New York: ACM Press, pp. 278–292.

Castagna, G., Ghelli, G. & Longo, G. (1995) A calculus for overloaded functions with subtyping. *Inf. Comput.* **117**(1), 115–135.

Coppo, M., Dezani-Ciancaglini, M. & Venneri, B. (1981) Functional characters of solvable terms. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* **27**, 45–58.

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

Davies, R. & Pfenning, F. (2000) Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*, pp. 198–208.

Dunfield, J. (2007) Refined typechecking with Stardust. In *Programming Languages meets Program Verification (PLPV '07)*, New York: ACM Press, pp. 21–32.

Dunfield, J. (2009) Greedy bidirectional polymorphism. In *ML Workshop*, pp. 15–26. Available at: `http://www.cs.queensu.ca/~jana/papers/poly/`

Dunfield, J. (2011) Untangling typechecking of intersections and unions. In *Proceedings of the 2010 Workshop on Intersection Types and Related Systems*, vol. 45 of *EPTCS*, pp. 59–70. arXiv:`1101.4428v1 [cs.PL]`.

Dunfield, J. (2012) Elaborating intersection and union types. In *International Conference on Functional Programming (ICFP)*, pp. 17–28. arXiv:`1206.5386 [cs.PL]`.

Dunfield, J. (2013) Twelf proofs accompanying this work, September. Available at: `http://www.cs.queensu.ca/~jana/intcomp-twelf-2013.tar`

Dunfield, J. & Krishnaswami, N. R. (2013) Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming (ICFP)*. arXiv:`1306.6032 [cs.PL]`.

Dunfield, J. & Pfenning, F. (2003) Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structures (FoSSaCS '03)*, pp. 250–266.

Dunfield, J. & Pfenning, F. (2004) Tridirectional typechecking. In *Principles of Programming Languages*, New York: ACM Press, pp. 281–292.

Freeman, T. & Pfenning, F. (1991) Refinement types for ML. In *Programming Language Design and Implementation*, New York: ACM Press, pp. 268–277.

Frisch, A., Castagna, G. & Benzaken, V. (2008) Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* **55**(4), 1–64.

Gentzen, G. (1969) Investigations into logical deduction. In *Collected Papers of Gerhard Gentzen*, Szabo, M. (ed), North-Holland: Amsterdam, pp. 68–131.

Hindley, J. R. (1984) Coppo–Dezani types do not correspond to propositional logic. *Theor. Comput. Sci.* **28**, 235–236.

Hindley, J. R. (1992) Types with intersection: An introduction. *Form. Asp. Comput.* **4**, 470–486.

Kfoury, A. J. & Wells, J. B. (2004) Principality and type inference for intersection types using expansion variables. *Theor. Comput. Sci.* **311**(1–3), 1–70.

Lopez-Escobar, E. G. K. (1985) Proof functional connectives. In *Methods in Mathematical Logic*, vol. 1130 of *Lecture Notes in Mathematics*, Springer: Berlin, pp. 208–221.

MacQueen, D., Plotkin, G. & Sethi, R. (1986) An ideal model for recursive polymorphic types. *Inf. Control* **71**, 95–130.

Milner, R., Tofte, M. Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. Cambridge: Massachusetts, USA, MIT Press.

Neergaard, P. M. & Mairson, H. G. (2004) Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *International Conference on Functional Programming (ICFP)*, pp. 138–149.

Pfenning, F. & Schürmann, C. (1999) System description: Twelf—a meta-logical framework for deductive systems. In *International Conference on Automated Deduction (CADE-16)*, pp. 202–206.

Pierce, B. C. (1991) *Programming with Intersection Types, Union Types, and Polymorphism*. Technical Report CMU-CS-91-106, Carnegie Mellon University.

Pierce, B. C. & Turner, D. N. (2000) Local type inference. *ACM Trans. Program. Lang. Syst.* **22**, 1–44.

Pottinger, G. (1980) A type assignment for the strongly normalizable lambda-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pages 561–577.

Rémy, D. (1989) Typechecking records and variants in a natural extension of ML. In *Principles of Programming Languages*, New York: ACM Press.

Reynolds, J. C. (1988) *Preliminary Design of the Programming Language Forsythe*. Technical Report CMU-CS-88-159, Carnegie Mellon University. `http://doi.library.cmu.edu/10.1184/OCLC/18612825`.

Reynolds, J. C. (1991) The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, vol. 526 of *LNCS*, Springer: Berlin, pp. 675–700.

Reynolds, J. C. (1996) *Design of the Programming Language Forsythe*. Technical Report CMU-CS-96-146, Carnegie Mellon University.

Tobin-Hochstadt, S. and Felleisen, M. (2008) The design and implementation of Typed Scheme. In *Principles of Programming Languages*, New York: ACM Press, pp. 395–406.

Turbak, F., Dimock, A., Muller, R. & Wells, J. B. (1997) Compiling with polymorphic and polyvariant flow types. In *International Workshop on Types in Compilation*. `Available at: http://cs.wellesley.edu/~fturbak/pubs/tic97.pdf`.

Twelf. (2012) Twelf wiki. `Available at: http://twelf.org/wiki/Main_Page`.

Wadler, P. and Blott, S. (1989) How to make *ad-hoc* polymorphism less *ad hoc*. In *Principles of Programming Languages*, New York: ACM Press, pp. 60–76.

Wells, J. B., Dimock, A., Muller, R. & Turbak, F. (2002) A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.* **12**(3), 183–227.

Wright, A. K. (1995) Simple imperative polymorphism. *LISP Symb. Comput.* **8**(4), 343–355.

Appendix A. Guide to the Twelf development

The Twelf proofs underlying the paper are available on the web:

`http://www.cs.queensu.ca/~jana/intcomp-2013.tar`	tar archive
`http://www.cs.queensu.ca/~jana/intcomp-2013/`	browsable files

All the lemmas and theorems in the paper were proved in Twelf version 1.7.1. The only caveat is that, to avoid the tedium of using nontrivial induction measures (Twelf only knows about subterm ordering), we use the *%trustme* directive to define *pacify*, yielding a blatantly unsound induction measure; see *base.elf*. All uses of this unsound measure can be found with

```
grep pacify *.elf
```

You can easily verify that in each case where *pacify* is used, the real inductive object is smaller according to either the standard depth (maximum path length) or weight (number of constructors, i.e., number of inference rules used) measures.

In any case, you will need to set Twelf's *unsafe* flag (*set unsafe true*) to permit the use of *%trustme* in the definition of *pacify*.

As usual, the Twelf configuration file is *sources.cfg*. We briefly describe the contents of each included *.elf* file:

- *base.elf* Generic definitions not specific to this paper.
- *syntax.elf* Source expressions *exp*, target terms *tm*, and types *ty*, covering much of Figures 2, 5 and 9.
- *is-value.elf* Which source expressions are values (Figure 2).
- *eval-contexts.elf* Evaluation contexts (Figure 2).
- *is-valuetm.elf* Which target terms are values (Figure 5).
- *typeof.elf* A system of rules for a version of $\Gamma \vdash e : A$ without subtyping. This system is related to the one in Figure 4 by Theorem 1 (*coerce.elf*).
- *typeof+sub.elf* The rules for $\Gamma \vdash e : A$ (Figure 4). Also defines subtyping *sub A B Coe CoeTyping*, corresponding to $A \leqslant B \hookrightarrow$ *Coe*. In the Twelf development, this judgment carries its own typing derivation (in the *typeof.elf* system, without subtyping) *CoeTyping*, which shows that the coercion *Coe* is well-typed.
- *sub-refl.elf* and *sub-trans.elf*: Reflexivity and transitivity of subtyping.
- *coerce.elf* Theorem 1: Given an expression well-typed in the system given in *typeof+sub.elf*, with full subsumption, coercions for function types can be inserted to yield an expression well-typed in the system of *typeof.elf*. Getting rid of subsumption makes the rest of the development easier.
- *elab.elf* Elaboration rules deriving $\Gamma \vdash e : A \hookrightarrow M$ from Figure 8.
- *typeof-elab.elf* Theorems 4 and 5.
- *typeoftm.elf* The typing rules deriving $G \vdash M : T$ from Figure 6.
- *elab-type-soundness.elf* Theorem 6.
- *step.elf* Stepping rules $e \rightsquigarrow e'$ (Figure 3).
- *step-eval-context.elf* Lemma 7 (stepping subexpressions in evaluation position).
- *steptm.elf* Stepping rules $M \mapsto M'$ (Figure 7).
- *tm-safety.elf* Theorems 2 and 3 (target type safety and determinism).
- *elab-union.elf*, *elab-sect.elf*, *elab-arr.elf* Inversion properties of elaboration for $\vee$, $\wedge$ and $\rightarrow$ (Lemmas 8, 9, and 10).
- *value-mono.elf* Value monotonicity of elaboration (Lemma 11).
- *consistency.elf* The main consistency result (Theorem 13) and its multistep version (Theorem 14).
- *summary.elf* Theorems 15 and 16, which are corollaries of earlier theorems.