Elaborating Intersection and Union Types

Jana Dunfield

Max Planck Institute for Software Systems
Kaiserslautern and Saarbrücken, Germany
jd169@queensu.ca

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.

Categories and Subject Descriptors F.3.3 [*Mathematical Logic and Formal Languages*]: Studies of Program Constructs—Type structure

Keywords intersection types

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. Logically, it roughly corresponds to conjunction[1]. Not surprisingly, then, intersection is remarkably versatile.

---

[1] In our setting, this correspondence is strong, as we will see in Sec. 2.

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

$$(\mathsf{even} * \mathsf{even} \rightarrow \mathsf{even}) \wedge (\mathsf{odd} * \mathsf{odd} \rightarrow \mathsf{even})$$
$$\wedge\, (\mathsf{even} * \mathsf{odd} \rightarrow \mathsf{odd}) \wedge (\mathsf{odd} * \mathsf{even} \rightarrow \mathsf{odd})$$

which satisfies the refinement restriction: all the intersected types refine a single simple type, bits $*$ bits $\rightarrow$ 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 Int.+ : int $*$ int $\rightarrow$ int and Real.+ : real $*$ real $\rightarrow$ real, we can easily define an overloaded addition operation by writing a merge:

$$\mathbf{val} + = \mathsf{Int.+} \,,,\, \mathsf{Real.+}$$

In our type system, this function + can be checked against the type (int $*$ int $\rightarrow$ int) $\wedge$ (real $*$ real $\rightarrow$ 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.
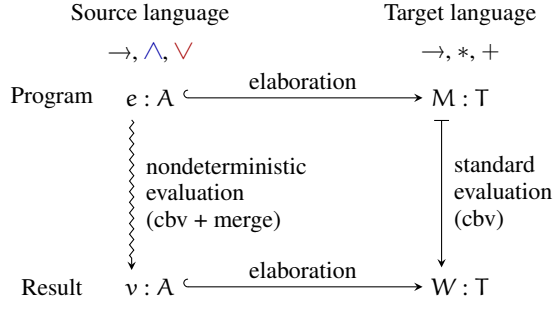
Figure 1. Elaboration and computation

1. An *encoding* phase that straightforwardly rewrites the program, for example, turning a multi-field record type into an intersection of single-field record types, and multi-field 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.

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 ($\leadsto^*$) 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) and are available on the web (Dunfield 2012). 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 behaviours expressed by the intersection. Remarkably, this merging construct also supports union elimina-

tions with two computationally distinct branches (unlike markers for union elimination in work such as Pierce (1991a)). 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 non-elaborating 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.

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 flavour. The intersection type in this paper is commutative ($A \wedge B = B \wedge A$) and idempotent ($A \wedge A = A$), following several 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[2] introduction and elimination rules for intersection to the (utterly standard) rules for 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.)

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

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

---

[2] 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.

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:

$$(A \to A) \wedge \underbrace{\big((A{\to}B{\to}C) \to (A{\to}B) \to A \to C\big)}_{\text{``D''}}$$

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 $A \to A$ is β-equivalent to $e_1 = \lambda x.\, x$, 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 $(A \to A) \wedge D$ 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 (throughout this paper, the subscript k ranges over $\{1, 2\}$):

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

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:

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

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

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

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 $(A \to A) * D$, 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 $A \,\widehat{\&}\, B$ as a proposition with some evidence of A that is also evidence of B—unlike $A \& B$, 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):

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

Coming up with a good elimination rule is trickier. A number of appealing rules are unsound; a sound, yet acceptably strong, rule is

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

This rule types an expression $\mathcal{E}[e_0]$—an evaluation context $\mathcal{E}$ with e_0 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 $f : (A \to A \to C) \wedge (B \to B \to C)$ and expression $e_0 : A \vee B$, where e_0 changes the contents pointed to by a reference of type $(A \vee B)$ ref, before returning the new 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 an A and the second e_0 to a B.

The evaluation context $\mathcal{E}$ need not be unique, which creates some difficulties for practical typechecking (Dunfield 2011). For further discussion of this rule, see Dunfield and Pfenning (2003).

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 $(A{\to}A) \wedge D$ (where D is the type of S) is uninhabited. In our setting, though, $(A{\to}A) \wedge D$ *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 int $\vee$ string. Introducing those unions to create the list is easy enough: use $\vee\text{I}_1$ for the ints and $\vee\text{I}_2$ for the strings. Now suppose we want to print a list element $x : \text{int} \vee \text{string}$, converting the ints to their string representation and leaving the strings alone. To do this, we need a merge; for example, given a function $g : (\text{int} \to \text{string}) \wedge (\text{string} \to \text{string})$ whose body contains a merge, use rule $\vee\text{E}$ on $g\ x$ with $\mathcal{E} = g\ []$ and $e_0 = x$.

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

4. Source Language

4.1 Source Syntax

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	$e ::= x \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{fix}\ x.\, e$ $\mid e_1 \,,\, e_2$
Source values	$v ::= x \mid () \mid \lambda x.\, e \mid v_1 \,,\, v_2$
Evaluation contexts	$\mathcal{E} ::= [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathcal{E} \,,\, e \mid e \,,\, \mathcal{E}$

Figure 2. Syntax of source types, contexts and expressions

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 $\top$ (which will be elaborated to unit), the usual function space $A \to B$, intersection $A \wedge B$ and union $A \vee B$. Values v are standard, but 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 (say) mutable references, it would not interact with them.

4.2 Source Operational Semantics

$$\boxed{e \rightsquigarrow e'} \quad \begin{array}{l} \text{Source expression } e \\ \text{steps to } e' \end{array} \quad \boxed{step\ E\ E'} \text{ in } step.elf$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 e_2 \rightsquigarrow e_1' e_2} \text{ step/app1} \qquad \frac{e_2 \rightsquigarrow e_2'}{v_1 e_2 \rightsquigarrow v_1 e_2'} \text{ step/app2}$$

$$\frac{}{(\lambda x.\, e) v \rightsquigarrow [v/x] e} \text{ step/beta}$$

$$\frac{}{\textbf{fix } x.\, e \rightsquigarrow [(\textbf{fix } x.\, e)/x] e} \text{ step/fix}$$

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

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

$$\frac{}{e \rightsquigarrow e,, e} \text{ step/split}$$

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

The source language operational semantics (Figure 3) is standard (call-by-value function application and a fixed point expression) except for the merge construct. This peculiar animal is a descendant of "demonic choice": 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 ('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. Worse, since the typing rule for merges ignores the unused part of the merge, $\rightsquigarrow$-reduction can produce expressions that have no type at all, or 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 $e \rightsquigarrow^* e'$ where e' elaborates to M'.

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 get inside the arrow $(A \wedge B) \to C$. For

flexibility, we'll incorporate a subtyping system that can conclude, for example, $A \to C \leq (A \wedge B) \to C$.

The logic of the subtyping rules (Figure 4, top) is taken straight from Dunfield and Pfenning (2003), so we only briefly give some intuition. Roughly, $A \leq B$ is sound if every value of type A can be treated as having type B. Under a subset interpretation, this would mean that $A \leq B$ is justified if the set of A-values is a subset of the set of B-values. For example, the rule $\wedge R\leq$, if interpreted set-theoretically, says that if $A \subseteq B_1$ and $A \subseteq B_2$ then $A \subseteq (B_1 \cap B_2)$.

It is easy to show that subtyping is reflexive and transitive; see $sub\text{-}refl.elf$ and $sub\text{-}trans.elf$. (Building transitivity into the structure of the rules makes it easy to derive an algorithm; an explicit transitivity rule would have premises $A \leq B$ and $B \leq C$, which involve an intermediate type B that does not appear in the conclusion $A \leq C$.)

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 $x : A \wedge B$. Operationally, all the coercions are identities; they serve only to "articulate" the type structure, making subsumption unnecessary.

Note that the coercion in rule $\vee L\leq$ is eta-expanded to allow $\vee E$ to eliminate the union in the type of x; as discussed later, 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 direct.

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: a use of the typing $e_0 : A$ within the derivation of $\mathcal{E}[e_0] : C$ is replaced by a derivations 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 (Section 9) follows a bidirectional version of them (where typechecking is decidable, given a few annotations, similar to Dunfield and Pfenning (2004)), where direct is *not* admissible. (A side benefit is that direct and $\vee E$ are similar enough that it can be helpful to do the direct case of a proof before tackling $\vee E$.)

Remark. Theorem 1, and all subsequent theorems, are proved only for expressions that are closed under the appropriate context, even though merge_k does not explicitly require that the unexamined subexpression be closed; Twelf does not support proofs about objects with unknown variables.

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 $A \leq B$, 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 $\wedge L_1\leq$ 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 $(e_{\text{coerce}}\ f)$

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

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

$$\frac{A_k \le B ::: e}{A_1 \wedge A_2 \le B ::: e}\ \wedge L_k\le \qquad \frac{A \le B_1 ::: e_1 \qquad A \le B_2 ::: e_2}{A \le B_1 \wedge B_2 ::: e_{1,,}\, e_2}\ \wedge R\le$$

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

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

$$\frac{}{\Gamma_1, x : A, \Gamma_2 \vdash x : A}\ \text{var} \qquad \frac{\Gamma \vdash e_k : A}{\Gamma \vdash e_{1,,}\, e_2 : A}\ \text{merge}_k \qquad \frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \mathbf{fix}\, x.\, e : A}\ \text{fix} \qquad \frac{}{\Gamma \vdash v : \top}\ \top I$$

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

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

$$\frac{\Gamma \vdash e_0 : A \qquad \Gamma, x : A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e_0] : C}\ \text{direct} \qquad \frac{\Gamma \vdash e : A_k}{\Gamma \vdash e : A_1 \vee A_2}\ \vee I_k \qquad \frac{\Gamma \vdash e_0 : A_1 \vee A_2 \quad \begin{array}{c}\Gamma, x_1 : A_1 \vdash \mathcal{E}[x_1] : C \\ \Gamma, x_2 : A_2 \vdash \mathcal{E}[x_2] : C\end{array}}{\Gamma \vdash \mathcal{E}[e_0] : C}\ \vee E$$

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

Figure 4. Source type system, with subsumption, non-elaborating

means that $[\,]$ is no longer in evaluation position. To handle these issues, 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 λ; 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. $\square$

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

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

Figure 5. Target types and terms

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

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 $M \mapsto M'$ is, likewise, standard; functions are call-by-value and products are strict. As usual, we write $M \mapsto^* M'$ for a sequence of zero or more $\mapsto$s.

Naturally, a type safety result holds:

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.) $\square$

And to calm any doubts about whether M might step to some other, not necessarily well-typed term:

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`. $\square$

6. Elaboration Typing

We elaborate 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 $+$.

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

$$\frac{}{G_1, x : T, G_2 \vdash x : T} \text{ typeoftm/ var}$$

$$\frac{G, x : T \vdash M : T}{G \vdash \mathbf{fix}\, x.\, M : T} \text{ typeoftm/ fix}$$

$$\frac{}{G \vdash () : \mathsf{unit}} \text{ typeoftm/ unitintro}$$

$$\frac{G, x : T_1 \vdash M : T_2}{G \vdash \lambda x.\, M : (T_1 \rightarrow T_2)} \text{ typeoftm/ arrintro}$$

$$\frac{G \vdash M_1 : T \rightarrow T' \qquad G \vdash M_2 : T}{G \vdash M_1\, M_2 : T'} \text{ typeoftm/ arrelim}$$

$$\frac{G \vdash M_1 : T_1 \qquad G \vdash M_2 : T_2}{G \vdash (M_1,\, M_2) : (T_1 * T_2)} \text{ typeoftm/ prodintro}$$

$$\frac{G \vdash M : (T_1 * T_2)}{G \vdash (\mathbf{proj}_k\, M) : T_k} \text{ typeoftm/ prodelim}_k$$

$$\frac{G \vdash M : T_k}{G \vdash (\mathbf{inj}_k\, M) : (T_1 + T_2)} \text{ typeoftm/ sumintro}_k$$

$$\frac{G \vdash M : T_1 + T_2 \qquad \begin{array}{c} G, x_1 : T_1 \vdash N_1 : T \\ G, x_2 : T_2 \vdash N_2 : T \end{array}}{G \vdash (\mathbf{case}\, M\, \mathbf{of}\, \mathbf{inj}_1\, x_1 \Rightarrow N_1 \mathbin{|} \mathbf{inj}_2\, x_2 \Rightarrow N_2) : T} \text{ typeoftm/ sumelim}$$

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

$\boxed{M \mapsto M'}$ Target term M steps to M' $\boxed{\begin{array}{c}\textit{steptm M M'} \\ \text{in } \textit{steptm.elf}\end{array}}$

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

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

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

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

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

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

Figure 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 8). 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).

The rest of this section discusses the elaboration rules and proves 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 various theorems.

Finally, Section 6.5 discusses whether we need a value restriction on $\wedge$I.

$$\begin{aligned} |\top| &= \mathsf{unit} \\ |A_1 \rightarrow A_2| &= |A_1| \rightarrow |A_2| \\ |A_1 \wedge A_2| &= |A_1| * |A_2| \\ |A_1 \vee A_2| &= |A_1| + |A_2| \end{aligned}$$

Figure 8. Type translation

6.1 Connecting Elaboration and Typing

Equivalence of elaboration and source typing: The non-elaborating type assignment system of Figure 4, minus sub, can be read off from the elaboration rules in Figure 9: simply drop the $\hookrightarrow \dots$ 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 straightforward induction on the given derivation; see $\textit{typeof-erase}$ in $\textit{typeof-elab.elf}$. $\square$

More interestingly, given $e : A$ 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* $\Gamma \vdash e : A \hookrightarrow M$.

Proof. By straightforward induction on the given derivation; see $\textit{elab-complete}$ in $\textit{typeof-elab.elf}$. $\square$

Elaboration produces well-typed terms: Any target term M produced by the elaboration rules has corresponding target type. In the theorem statement, we assume the obvious translation $|\Gamma|$, e.g. $|x : \top, y : \top \vee \top| = x : |\top|, y : |\top \vee \top| = x : \mathsf{unit}, y : \mathsf{unit} + \mathsf{unit}$).

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 $\textit{elab-type-soundness.elf}$. $\square$

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

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

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

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

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

$$\frac{\Gamma \vdash e_0 : A \hookrightarrow M_0 \qquad \Gamma, x : A \vdash \mathcal{E}[x] : C \hookrightarrow N}{\Gamma \vdash \mathcal{E}[e_0] : C \hookrightarrow (\lambda x. \, N) M_0} \; \text{direct} \qquad \frac{\Gamma \vdash e_0 : A_1 \vee A_2 \hookrightarrow M_0 \qquad \begin{array}{l} \Gamma, x_1 : A_1 \vdash \mathcal{E}[x_1] : C \hookrightarrow N_1 \\ \Gamma, x_2 : A_2 \vdash \mathcal{E}[x_2] : C \hookrightarrow N_2 \end{array}}{\Gamma \vdash \mathcal{E}[e_0] : C \hookrightarrow \mathbf{case} \; M_0 \; \mathbf{of} \; \mathbf{inj}_1 \; x_1 \Rightarrow N_1 \mid \mathbf{inj}_2 \; x_2 \Rightarrow N_2} \; \vee\text{E}$$

Figure 9. Elaboration typing rules

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 $\wedge\text{E}_2$, 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 $\wedge\text{I}$, 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 behaviour. 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 ($step\text{-}eval\text{-}context$) that $e \rightsquigarrow e'$ implies $\mathcal{E}[e] \rightsquigarrow \mathcal{E}[e']$. See $step*eval\text{-}context$ in $step\text{-}eval\text{-}context.elf$. $\square$

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. For intersections, the property is slightly more complicated: given an expression of intersection type that elaborates to a pair, we can step the expression to get something that elaborates to the components of the pair. 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 derivation of $\Gamma \vdash e : C \hookrightarrow M$. The only possible cases are merge_k and $\vee\text{I}_k$. See $elab\text{-}inl$ and $elab\text{-}inr$ in $elab\text{-}union.elf$. $\square$

Lemma 9 (Intersections/Pairs)**.**
If $\Gamma \vdash e : A_1 \wedge A_2 \hookrightarrow (M_1 \,,\, M_2)$ then there exist e_1' and e_2' such that

(1) $e \rightsquigarrow^* e_1'$ and $\Gamma \vdash e_1' : A_1 \hookrightarrow M_1$, and
(2) $e \rightsquigarrow^* e_2'$ and $\Gamma \vdash e_2' : A_2 \hookrightarrow M_2$.

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

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 $\to\text{I}$ and merge. See $elab\text{-}arr.elf$. $\square$

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 $e \rightsquigarrow^* v$ where $\Gamma \vdash v : A \hookrightarrow W$.*

Proof. By induction on the given derivation.

The most interesting case is for $\wedge\text{I}$, where we apply the induction hypothesis to each premise (yielding v_1', v_2' such that $e \rightsquigarrow^* v_1'$ and $e \rightsquigarrow^* v_2'$), apply the 'step/split' rule to turn e into $(e \,{}_{,,}\, e)$, and use the 'step/merge1' and 'step/merge2' rules to step each part of the merge, yielding $v_1' \,{}_{,,}\, v_2'$, which is a value.

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 $value\text{-}mono.elf$. $\square$

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. As usual, Twelf gives us this substitution lemma for free. $\square$

6.3 Consistency

This theorem 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 $\mathbf{proj}_1 \; (W_1 \,,\, W_2)$ and steps to W_1, there is nothing to do in e because the projection corresponds to the *implicit* elimination in rule $\wedge\text{E}_1$;

- 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 $e \rightsquigarrow e'$ 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 $e \rightsquigarrow^* e'$ 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., $e \rightsquigarrow^* e_1'$ and $\cdot \vdash e_1' : A_1 \hookrightarrow M_1'$. By 'step/split', $e \rightsquigarrow e,, e$. Repeatedly applying 'step/merge1' gives $e,, e \rightsquigarrow^* e_1',, e$.
  For typing, apply merge_1 with premise $\cdot \vdash e_1' : A_1 \hookrightarrow M_1'$ and 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$_k$:**
$$\mathcal{D} :: \frac{\cdot \vdash e : A_1 \wedge A_2 \hookrightarrow M_0}{\cdot \vdash e : A_k \hookrightarrow \mathbf{proj}_k M_0}$$

  If $\mathbf{proj}_k M_0 \mapsto \mathbf{proj}_k M_0'$ 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 $e \rightsquigarrow^* e_k'$ and $\Gamma \vdash e_k' : A_k \hookrightarrow W_k$.

- **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 $\cdot \vdash e' : A$. By rule 'step/unmerge', $e_1,, e_2 \rightsquigarrow e_k$. Therefore $e_1,, e_2 \rightsquigarrow^* e'$.

- **Case $\rightarrow$E:**
$$\mathcal{D} :: \frac{\cdot \vdash e_1 : A \rightarrow B \hookrightarrow M_1 \qquad \cdot \vdash e_2 : A \hookrightarrow M_2}{\cdot \vdash e_1 e_2 : B \hookrightarrow M_1 M_2}$$

  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 ::	$\cdot \vdash e_1 : A \rightarrow B \hookrightarrow \lambda x. M_0$	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$'$) $\square$

Theorem 14 (Multi-step 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`. $\square$

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). $\square$

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 (multi-step consistency) and 4. $\square$

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

Davies and Pfenning (2000) showed that the then-standard intersection introduction (that is, our $\wedge$I) was unsound in a call-by-value semantics in the presence of effects (specifically, mutable references). Here is an example (modeled on 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 pos $\leq$ nat.

$$\begin{aligned}&\mathbf{let}\ r = (\mathbf{ref}\ 1) : (\text{nat ref}) \wedge (\text{pos ref})\ \mathbf{in}\\ &\quad r := 0;\\ &\quad (!r) : \text{pos}\end{aligned}$$

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

Davies and Pfenning proposed, analogously to ML's value restriction on $\forall$-introduction, an $\wedge$-introduction rule that only types values v. This rule is sound with mutable references:

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

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

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

Thus, the example elaborates to

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

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 $\wedge \mathrm{I}$ thus appears sound in our setting. Being *sound* is not the same as being *useful*, though; such behaviour is less than intuitive, as we discuss in the next section.

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 they would have overlapping types—types with 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_k and $\wedge \mathrm{I}$.

According to the elaboration rules, $0,, 1$ (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 behaviour than actual randomness, but hardly helpful to the programmer. Perhaps even more confusingly, suppose we are checking $0,, 1$ against $\mathsf{pos} \wedge \mathsf{nat}$, where pos and nat are as in Section 6.5. Our implementation would elaborate $0,, 1$ to $(1,\ 0)$, but $1,, 0$ 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 this seems restrictive; also, 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 might 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). Implementing this seems straightforward, though it would slow typechecking since we could not skip over e_2 when e_1 checks in $e_1,, e_2$.

Leaving merges aside, the mere fact that $\wedge \mathrm{I}$ elaborates the expression twice creates problems with mutable references, as we saw in Section 6.5. For this, we could revive the value restriction in $\wedge \mathrm{I}$, 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 fixed set of built-in operations; it is not possible to write a single `square` function, as we do in Figure 10. Despite its appearance, (*[**val** square : ...]*) is not a comment but an annotation used to guide our bidirectional typechecker (this syntax, inherited from Stardust, was intended for compatibility with SML compilers, which saw these annotations as comments and ignored them).

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

```
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`

Figure 10. Example of overloading

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 `{fld : A}` with introduction form `{fld= e}` and the usual eliminations (explicit projection and pattern matching). Once this is done, the multi-field record type `{fld1 : A₁, fld2 : A₂}` is simply `{fld1 : A₁}` $\wedge$ `{fld2 : A₂}`, and the corresponding intro form is a merge: `{fld1= A₁},, {fld2= A₂}`. More standard concrete syntax, such as `{fld1= A₁, fld2= A₂}`, can be handled trivially during parsing.

With subtyping on intersections, we get the desired behaviour 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
```

and use `int_or_string list`s. 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 $(\mathsf{int} \to \mathsf{string}) \wedge (\mathsf{string} \to \mathsf{string})$).

```
(*[ 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)
```

Figure 11. Example of flexible multi-field records

```
datatype 'a list = nil | :: of 'a * 'a list

type dyn = int ∨ real ∨ string

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

(*[ val hetListToString : dyn list → 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
```

Figure 12. Example of heterogeneous data

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 ∨ string
```

Elaboration transforms programs with `int_union_string` into programs with `int_or_string`.

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

9. Implementation

Our implementation is faithful to the spirit of the elaboration rules above, but is substantially richer. It is based on Stardust, a type-checker for a subset of core Standard ML with support for inductive datatypes, products, intersections, unions, refinement types and indexed types (Dunfield 2007), extended with support for (first-class) polymorphism (Dunfield 2009). We do not yet support all these features; support for first-class polymorphism looks hardest, since Standard ML compilers cannot even handle higher-rank predicative polymorphism. Elaborating programs that use ML-style prenex polymorphism should work, but we currently lack any proof or even significant testing 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 multi-field records (Section 8.2), and inductive datatypes (if their constructors are not of intersection type, though they can take intersections and unions as argument; removing this restriction is a high priority).

9.1 Bidirectional Typechecking

Our implementation uses *bidirectional typechecking* (Pierce and Turner 2000; Dunfield and Pfenning 2004; Dunfield 2009), an increasingly common technique in advanced type systems; see Dunfield (2009) for references. 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 more localized error messages.

Bidirectional typechecking does need more type annotations. However, by following the approach of Dunfield and Pfenning (2004), annotations are never needed except on redexes. The present implementation 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:

$$\Gamma \vdash e \Leftarrow A \qquad e \text{ checks against known type } A$$
$$\Gamma \vdash e \Rightarrow A \qquad e \text{ synthesizes type } A$$

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).

Syntactically speaking, crafting a bidirectional type system from a type assignment system (like the one in Figure 4) is a matter of taking the colons in the $\Gamma \vdash e : A$ judgments, and replacing some with "$\Leftarrow$" and some with "$\Rightarrow$". Except for $\mathsf{merge_k}$, our typing rules can all be found in Dunfield and Pfenning (2004), who argued that introduction rules should check and elimination rules should synthesize. (Parametric polymorphism muddies this picture, but see Dunfield (2009) for an approach used by our implementation.) For functions, this leads to the bidirectional rules

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

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):

$$\frac{\Gamma \vdash e_k \Leftarrow A}{\Gamma \vdash e_1{}_{,,}\, e_2 \Leftarrow A}$$

Since it can be inconvenient to annotate merges, we also implement synthesis rules, including one that can synthesize an intersection.

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

Given a bidirectional typing derivation, it is generally easy to show that a corresponding type assignment exists: replace all "⇒" and "⇐" with ":" (and erase explicit type annotations from the expression).

9.2 Performance

Intersection typechecking is PSPACE-hard (Reynolds 1996). In practice, we elaborate the examples in Figures 10, 11 and 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 labour of backtracking remain the same.

To scale 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 2-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 may require 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. Work on union types began later (MacQueen et al. 1986); Barbanera et al. (1995) is a key paper on type assignment for unions.

Forsythe. In the late 1980s[3], Reynolds invented Forsythe (Reynolds 1996), the first practical programming language based on intersection types. 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 can only be used unambiguously. For instance, a record and a function can be merged, but two functions cannot (actually they can, but the second phrase p_2 overrides the first). Forsythe does not have union types.

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 nat $\rightarrow$ nat and one of type pos $\rightarrow$ 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.

Pierce's work. The earliest reference I know for the idea of compiling intersection to product is Pierce (1991b): "a language with intersection types might even provide two different object-code sequences for the two versions of + [for int and for real]" (p. 11). Pierce also developed a language with union types, including a term-level construct to explicitly eliminate them (Pierce 1991a). 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 $\wedge I$.

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 dynamic typing's transparency and static typing's guarantees 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 provide 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 behaviours 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

---

[3] The citation year 1996 is the date of the revised description of Forsythe; the core ideas are found in Reynolds (1988).

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 non-values 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.

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 (pos $\rightarrow$ neg) $\wedge$ (neg $\rightarrow$ pos) to a pair of functions seems pointless (unless it can somehow facilitate optimizations in the compiler). It will probably be necessary to have "refinement" and "unrestricted" versions of the intersection and union type constructors, at least during elaboration; it may be feasible to hide this distinction at the source level.

Acknowledgments

References

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

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119:202–230, 1995.

Robert Cartwright and Mike Fagan. Soft typing. In *Programming Language Design and Implementation*, pages 278–292, 1991.

Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

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

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

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

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

Jana Dunfield. Greedy bidirectional polymorphism. In *ML Workshop*, pages 15–26, 2009. http://www.cs.queensu.ca/~jana/papers/poly/.

Jana Dunfield. Untangling typechecking of intersections and unions. In *2010 Workshop on Intersection Types and Related Systems*, volume 45 of *EPTCS*, pages 59–70, 2011. arXiv:1101.4428v1[cs.PL].

Jana Dunfield. Twelf proofs accompanying this paper, March 2012. http://www.cs.queensu.ca/~jana/intcomp.tar or http://www.cs.queensu.ca/~jana/intcomp/.

Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FoSSaCS '03)*, pages 250–266, 2003.

Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In *Principles of Programming Languages*, pages 281–292, 2004.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991.

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

J. Roger Hindley. Coppo-Dezani types do not correspond to propositional logic. *Theoretical Computer Science*, 28:235–236, 1984.

J. Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4:470–486, 1992.

Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1–3):1–70, 2004.

David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71: 95–130, 1986.

Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *ICFP*, pages 138–149, 2004.

Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Int'l Conf. Automated Deduction (CADE-16)*, pages 202–206, 1999.

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

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

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Prog. Lang. Syst.*, 22:1–44, 2000.

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

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

John C. Reynolds. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 675–700. Springer, 1991.

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

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types. In *Int'l Workshop on Types in Compilation*, 1997.

Twelf. Twelf wiki, 2012. http://twelf.org/wiki/Main_Page.

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Principles of Programming Languages*, pages 60–76, 1989.

J.B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Functional Programming*, 12(3):183–227, 2002.