A Paradigm Shift in Program Analysis and Transformation via Intersection and Union Types

Joe Wells
Heriot-Watt University
Computing & Elec. Eng. Dept.
http://www.cee.hw.ac.uk/~jbw

March 28, 2000

Overview

- The Appeal of Types.
- Type Polymorphism: What and Why.
- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)
- Future Work and Conclusion.

Criteria for Type Systems

Types reduce errors; type systems with certain properties do this better:

- *Strong typing*: Typable programs are *safe*: They have a well-defined meaning and, if they fail, do so in a well-defined way. (E.g., *not* C, C++, Pascal, etc.)

- *Static typing*: Type errors are discovered at compile-time rather than at run-time. (E.g., *not* LISP, or Scheme (debatable).)

It is burdensome to actually have to supply types, but this problem can be mitigated by this property:

- *Implicit typing*: Most or all type information is determined by the compiler. (E.g., *not* Java.) Benefit: avoids difficulty of supplying types explicitly.

Application: Type-Directed Compilation

- Early compilers for languages with strong static type systems (e.g., Haskell, ML) discarded types after type checking.
- Starting mid 90's, some compilers have used a *typed intermediate language* (TIL) [Harper and Morrisett, 1995; Peyton Jones, 1996]. Benefits: *type-directed compilation* can choose more efficient representations, helps detect compiler bugs. Peyton Jones:

  > "It is difficult to write an incorrect but well-typed program transformation."

- Very recent ML compilers have also used a *typed assembly language* (TAL) [Morrisett et al., 1998]. Benefits: claims of type safety more meaningful, certifying mobile machine code.

Application: Types for Program Analysis

Starting in late '80s, another less traditional use of types is to structure and guide analyses of properties other than type safety, such as:

- *Flow*, e.g., Heintze [1995]; Jagannathan et al. [1997]; Palsberg and O'Keefe [1995]. More later in talk.
- *Effects* [Lucassen and Gifford, 1988; Talpin and Jouvelot, 1994],

  strictness [Jensen, 1998], and *binding-time* [Thiemann, 1997]. These analyses support optimizations.
- *Security*, e.g., Abadi, Banerjee, Heintze, and Riecke [1999]. Understanding information flow is essential here.

All of these analyses are harder for *higher-order* languages.

Overview

- The Appeal of Types.
- Type Polymorphism: What and Why.
- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)
- Future Work and Conclusion.

Type Polymorphism

An important feature mitigating type system inflexibility is *type polymorphism*, which:

- Allows a program fragment to be viewed in different ways, depending on where its output is used or where its inputs come from.
- Is *essential* for code reuse [Reynolds, 1974] and abstract data types [Mitchell and Plotkin, 1988].
- Is traditionally treated formally using universal ($\forall$) quantifiers [Girard, 1972] or existential ($\exists$) quantifiers and/or by a notion of subtyping ($\sigma \leq \tau$).

Polymorphism Example with Universal Quantifiers

$$\texttt{val swap}^{\forall\alpha,\beta.(\alpha\times\beta)\to(\beta\times\alpha)} = (\texttt{fn}\ (\texttt{x}^{\alpha}, \texttt{y}^{\beta}) \Rightarrow (\texttt{y}^{\beta},\ \texttt{x}^{\alpha}));$$

$$\texttt{val pair1}^{\texttt{int}\times\texttt{bool}} = (1, \texttt{true});$$

$$\texttt{val pair2}^{\texttt{real}\times\texttt{real}} = (2.7, 9.9);$$

$$(\texttt{swap}^{(\texttt{int}\times\texttt{bool})\to(\texttt{bool}\times\texttt{int})}\ \texttt{pair1},$$
$$\ \texttt{swap}^{(\texttt{real}\times\texttt{real})\to(\texttt{real}\times\texttt{real})}\ \texttt{pair2});$$

- Implicit typing discovers the types automatically [Damas and Milner, 1982] (at least for the above example).

- In body of polymorphic function, the usage types are hidden behind type variables.

Polymorphism Example with Existential Quantifiers

$$\texttt{val closure1}^{\texttt{int}\times(\texttt{int}\to\texttt{bool})} = (5,\ (\texttt{fn x} \Rightarrow \texttt{x} > 1));$$

$$\texttt{val closure2}^{\texttt{bool}\times(\texttt{bool}\to\texttt{bool})} = (\texttt{true},\ (\texttt{fn x} \Rightarrow \texttt{not x}));$$

$$\begin{aligned}\texttt{val closure} = \texttt{if b then } & \texttt{closure1}^{\exists\alpha.\alpha\times(\alpha\to\texttt{bool})}\\ \texttt{else } & \texttt{closure2}^{\exists\alpha.\alpha\times(\alpha\to\texttt{bool})};\end{aligned}$$

$$\texttt{val result}^{\texttt{bool}} = (\#2\ \texttt{closure})^{\alpha\to\texttt{bool}}(\#1\ \texttt{closure})^{\alpha};$$

- Dual of universal quantifier.

- Usage site does not know source types.

An Alternative: Intersection and Union Types

- Type polymorphism by *listing* usage types [Coppo et al., 1980; Barbanera et al., 1995].

- Called "intersection" (and "union") types because, if we think of σ and τ as denoting sets of program fragments, the type $\sigma \cap \tau$ denotes exactly the intersection of the sets denoted by σ and τ.

- Example comparing intersection and universal types:

$$\begin{array}{ll} \text{intersection types:} & (\mathtt{fn\ x} \Rightarrow \mathtt{x})^{(\mathtt{int}\to\mathtt{int})\cap(\mathtt{bool}\to\mathtt{bool})} \\ \text{universal types:} & (\mathtt{fn\ x} \Rightarrow \mathtt{x})^{\forall\alpha.\alpha\to\alpha} \end{array}$$

  The example intersection type is semantically like $\forall\alpha \in \{\mathtt{int}, \mathtt{bool}\}.\alpha \to \alpha$, but there are significant practical differences.

Polymorphism Example with Intersection Types

$$\mathtt{val\ swap}^{\left(\bigcap\substack{(\mathtt{int}\,\times\,\mathtt{bool})\,\to\,(\mathtt{bool}\,\times\,\mathtt{int})\\(\mathtt{real}\,\times\,\mathtt{real})\,\to\,(\mathtt{real}\,\times\,\mathtt{real})}\right)} = (\mathtt{fn}\ (\mathtt{x}^{\substack{\mathtt{int}\\\mathtt{real}}}, \mathtt{y}^{\substack{\mathtt{bool}\\\mathtt{real}}}) \Rightarrow (\mathtt{y}^{\substack{\mathtt{bool}\\\mathtt{real}}},\ \mathtt{x}^{\substack{\mathtt{int}\\\mathtt{real}}}));$$

$$\mathtt{val\ pair1}^{\mathtt{int}\times\mathtt{bool}} = (1, \mathtt{true});$$

$$\mathtt{val\ pair2}^{\mathtt{real}\times\mathtt{real}} = (2.7, 9.9);$$

$$(\mathtt{swap}^{(\mathtt{int}\times\mathtt{bool})\to(\mathtt{bool}\times\mathtt{int})}\ \mathtt{pair1},$$
$$\ \mathtt{swap}^{(\mathtt{real}\times\mathtt{real})\to(\mathtt{real}\times\mathtt{real})}\ \mathtt{pair2});$$

- All types can be discovered automatically [van Bakel, 1993; Jim, 1996; Kfoury and Wells, 1999]. (Also Ronchi Della Rocca [1988].)

- Exposes usage types throughout.

Polymorphism Example with Union Types

$$\texttt{val closure1}^{\texttt{int}\times(\texttt{int}\to\texttt{bool})} = (5,\ (\texttt{fn x} \Rightarrow \texttt{x} > 1));$$

$$\texttt{val closure2}^{\texttt{bool}\times(\texttt{bool}\to\texttt{bool})} = (\texttt{true},\ (\texttt{fn x} \Rightarrow \texttt{not x}));$$

$$\begin{aligned}\texttt{val closure} = \texttt{if b then closure1}&^{\left(\bigcup\begin{smallmatrix}\texttt{int}\times(\texttt{int}\to\texttt{bool})\\ \texttt{bool}\times(\texttt{bool}\to\texttt{bool})\end{smallmatrix}\right)}\\ \texttt{else closure2}&^{\left(\bigcup\begin{smallmatrix}\texttt{int}\times(\texttt{int}\to\texttt{bool})\\ \texttt{bool}\times(\texttt{bool}\to\texttt{bool})\end{smallmatrix}\right)};\end{aligned}$$

$$\texttt{val result}^{\texttt{bool}} = (\#2\ \texttt{closure})^{\begin{smallmatrix}\texttt{int}\to\texttt{bool}\\ \texttt{bool}\to\texttt{bool}\end{smallmatrix}}(\#1\ \texttt{closure})^{\begin{smallmatrix}\texttt{int}\\ \texttt{bool}\end{smallmatrix}};$$

- Dual of intersection types.
- Exposes usage types throughout.

More on Intersection/Union Types

- Programmers can not be expected to supply intersection/union types; they must be derived automatically.

- Types more closely approximate actual behavior. Slogan:

  Intersection/union types for *analysis*;
  universal/existential types for *specification*.

Overview

- The Appeal of Types.

- Type Polymorphism: What and Why.

- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)

- Future Work and Conclusion.

Principal Typings: What

- A *typing* $T = (A \vdash \tau)$ is a set of *type assumptions* A and a *result type* τ.
- A typing $T = (A \vdash \tau)$ determines a set $[\![T]\!]$ of all the program fragments M such that $A \vdash M : \tau$ is provable ("M is well typed with result type τ under the type assumptions A").
- A typing T_1 is *more general* than typing T_2 iff $[\![T_1]\!] \subset [\![T_2]\!]$.

  (T_1 provides more information about $M \in [\![T_1]\!]$, allowing it to be used in more contexts.)
- Typing T is *principal* for fragment M iff T is a most general typing for M.

Principal Typings: Why

- Jim [1996]:

  "The intuition is, a principal typing EXPECTS LESS of its free variables, and PROVIDES MORE [for its result] than any other typing judgment."

- Principal typings allow *compositional* type analysis:
  - The type analysis for a fragment uses only the *results* for its subfragments, which can be analyzed independently in any order.
  - Helps with separate compilation and modularity.
  - Helps in making a complete/terminating analysis algorithm.

Universal Quantifiers Prevent Principal Typings

Counterexample for Girard and Reynold's System F, well known extensions of System F such as F+η, and the finite-rank restrictions Λ_k for $k \geq 3$:

- Suppose $T = (A \vdash \tau)$ is a principal typing for the program fragment $(\mathtt{x}\,\mathtt{x})$.
- Let σ be the result of erasing all occurrences of "$\forall$" from $A(\mathtt{x})$. Let $B(\mathtt{x}) = (\forall.(\sigma \to \beta))$ where β is fresh. Let $T' = (B \vdash \rho)$ for some ρ.
- Then $(\mathtt{x}\,\mathtt{x}) \in [\![T']\!]$ and there exists an M such that $((\mathtt{fn}\ \mathtt{y} \Rightarrow \mathtt{x}\,\mathtt{x})\,M) \in [\![T]\!] \setminus [\![T']\!]$, contradicting the claim that T is principal for $(\mathtt{x}\,\mathtt{x})$.

  The term M built by methods of Wells [1994, 1996a,b, 1999] "tests" the size of the type of $\mathtt{x}$ and fails if the type is too big.

Impact of Lack of Principality

- Difficulty in separate compilation. The elaborate compilation manager (CM) facility of ML is needed partly due to this problem.

- Difficulty in making type systems more flexible than the restrictive Hindley/Milner system.
  - Many desire Scheme or Smalltalk-like flexibility and analyses beyond safety-checking need greater precision.
  - But no computable/terminating type analysis algorithm exists for extensions such as System F [Wells, 1994], the rank-k restriction of F for $k \geq 3$ [Kfoury and Wells, 1994], System F with subtyping [Wells, 1996a], F_ω [Urzyczyn, 1997]. Even some apparently simple subtyping relations are undecidable [Pierce, 1994; Tiuryn and Urzyczyn, 1996; Wells, 1995].

Intersection Types to the Rescue!

- Systems with intersection types *do* have principality [Coppo et al., 1980; Ronchi Della Rocca and Venneri, 1984; van Bakel, 1993; Jim, 1996], although the formulation has either been extremely complicated or for quite restricted systems until recently.

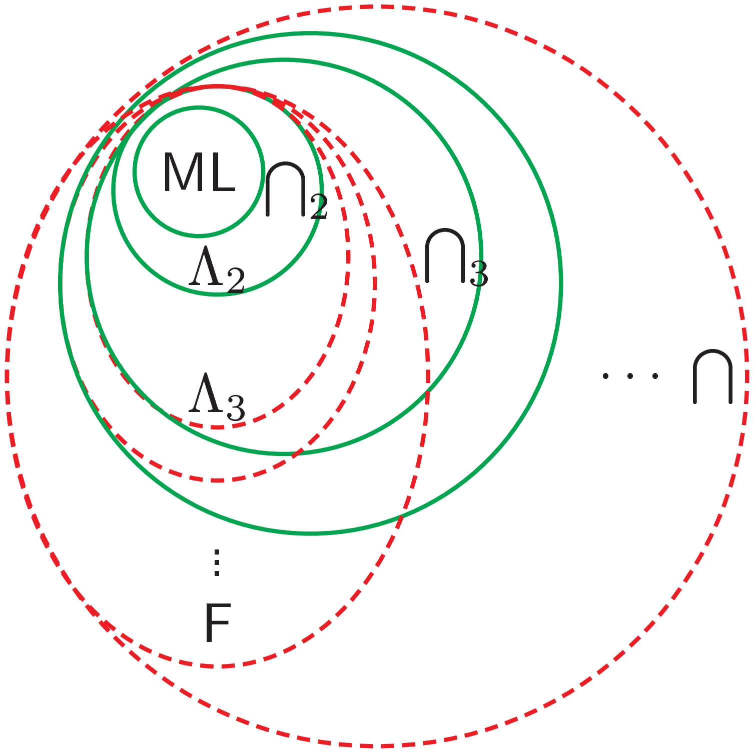
- There is now an understandable principal typing algorithm for the full, unrestricted system of intersection types using the new technology of *expansion variables* [Kfoury and Wells, 1999].

  The implementation is mostly finished. (Unfortunately the student escaped ("graduated") before finishing.)

- Intuition why it works: Semantically, intersection is a glb and union is a lub.

Decidability of Typability for Various Systems

F: System F.
Λ_k: rank-k System F.
$\bigcap$: intersection types.
$\bigcap_k$: rank-k of $\bigcap$.
Decidable.
Undecidable.



(Asymptotic complexity now known [Kfoury, Mairson, Turbak, and Wells, 1999].)

Example Typable only with Intersection Types

```
fun self_apply2 z  ⇒ (z z) z;
fun apply f x ⇒ f x;
fun reverse_apply y g ⇒ g y;
fun id w ⇒ w;
(self_apply2 apply not true,
 self_apply2 reverse_apply id false not);
```

- The example *safely* computes (`false`, `true`).
- Urzyczyn [1997] proved this example is not typable in F_ω, considered the most powerful type system with universal quantifiers.
- The example is typable in the rank-3 restriction of intersection types.

Overview

- The Appeal of Types.
- Type Polymorphism: What and Why.
- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)
- Future Work and Conclusion.

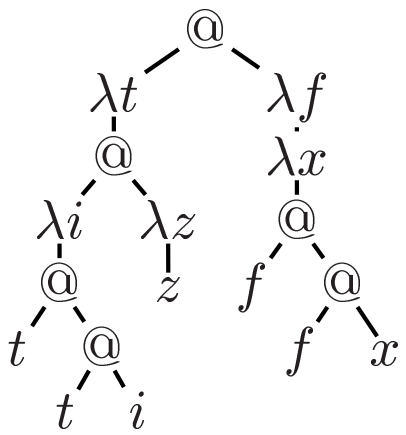
An Example Program

This ML program:

```
fun twice f x = f (f x);
fun id z = z;
twice (twice id);
```

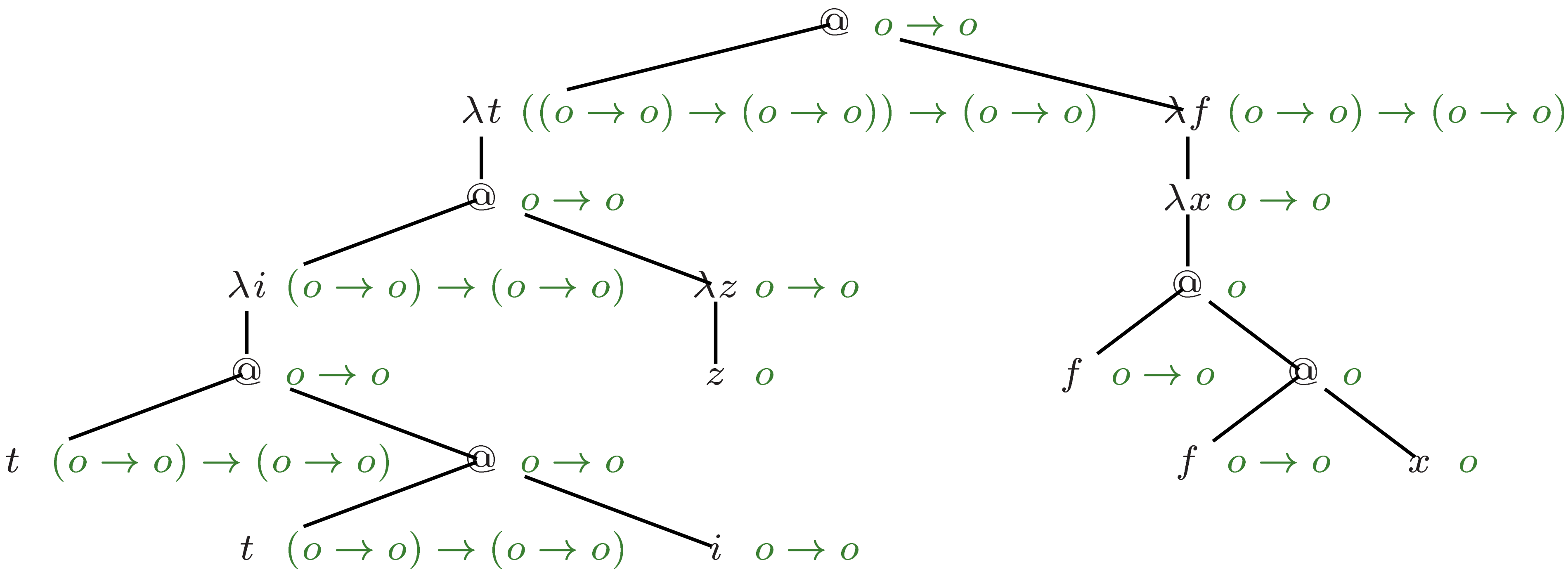
is the same as this λ-term: $(\lambda t.(\lambda i.t(ti))(\lambda z.z))(\lambda f.\lambda x.f(fx))$

which can be drawn as this tree:



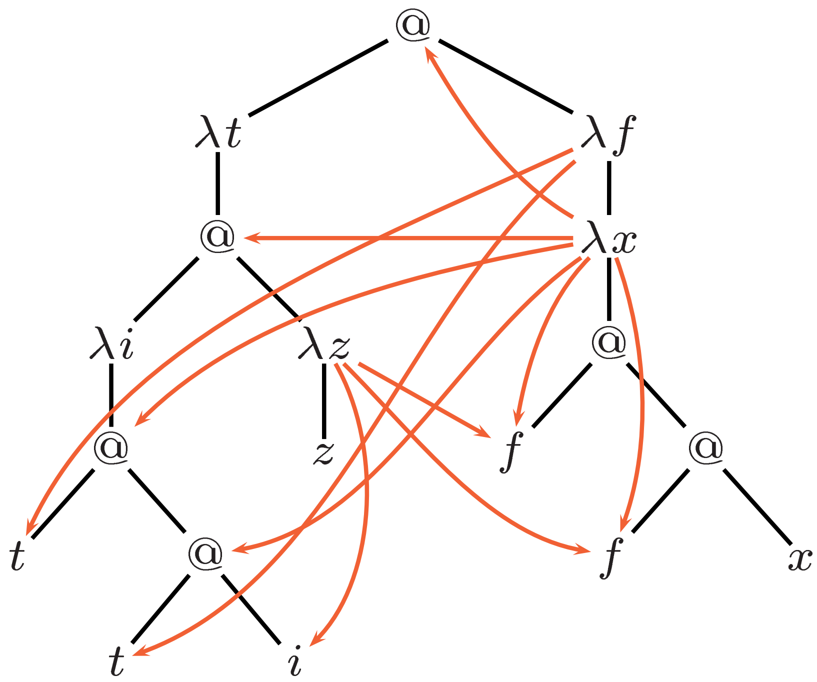
Example: Types

Our example analyzed using the simply typed λ-calculus:



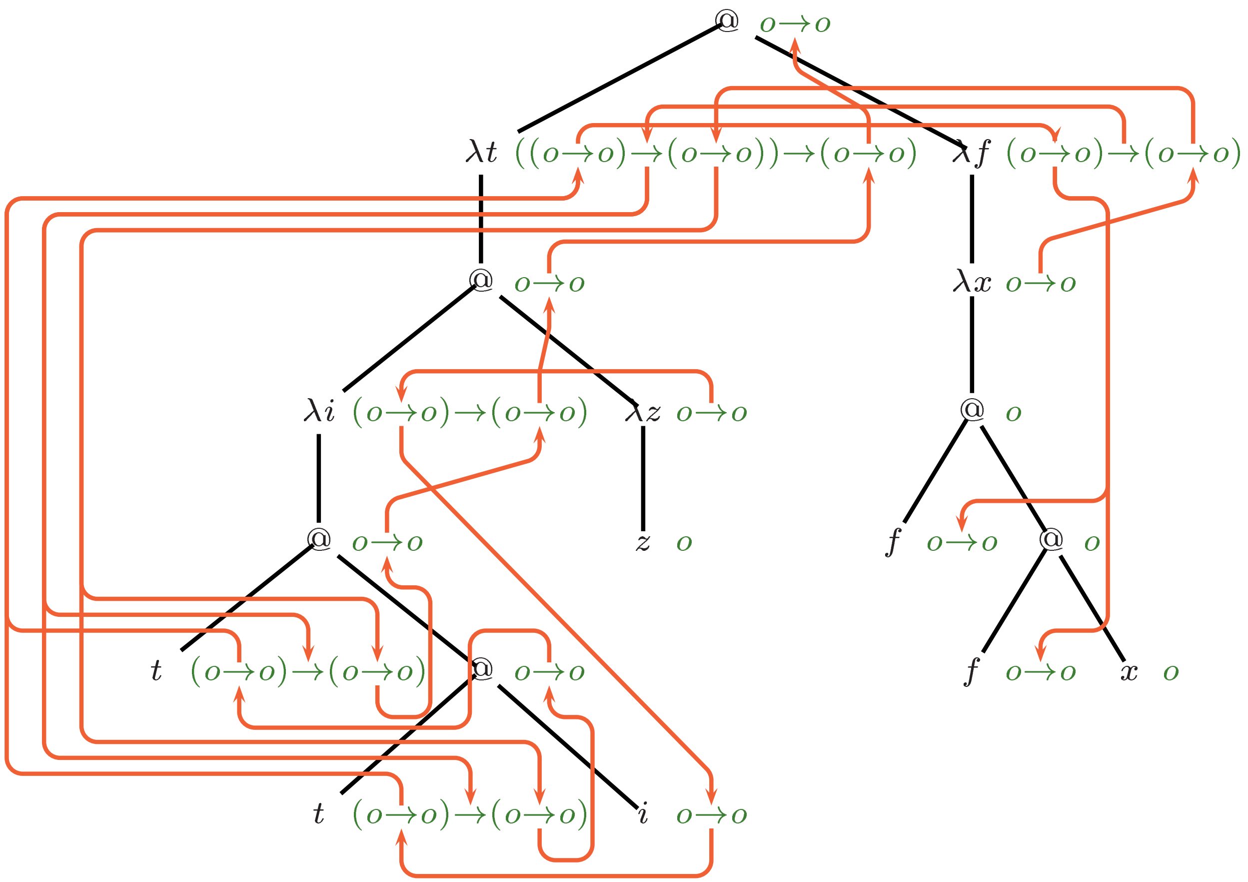
Example: Flow

Our example analyzed using 0CFA [Shivers, 1991]:



Example: Types and Flow

Illustrating how the type and flow analyses are intertwined:



Type Analysis *Is* Flow Analysis

- The *higher-order flow graph* (example on previous page) reveals how type and flow information is intertwined [Heintze, 1995; Palsberg and O'Keefe, 1995].

- Universal/existential quantifiers interfere with the higher-order flow graph style of unifying flow and type information — the quantified type variables hide information.

- Intersection/union types, by exposing usage types, are less painful to use for flow analysis [Wells et al., 1997; Palsberg and Pavlopoulou, 1998; Banerjee, 1997].

- Banerjee [1997] showed the principality of intersection types means flow analysis does *not* need the whole program!

Overview

- The Appeal of Types.
- Type Polymorphism: What and Why.
- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)
- Future Work and Conclusion.

Universal/Existential Types and (Un)Boxing

- Remember the polymorphic function example:

$$\texttt{val swap}^{\forall\alpha,\beta.(\alpha\times\beta)\to(\beta\times\alpha)} = (\texttt{fn}\ (\texttt{x}^{\alpha}, \texttt{y}^{\beta}) \Rightarrow (\texttt{y}^{\beta},\ \texttt{x}^{\alpha}));$$

  Because the types do not tell the representation of α and β, all types which α and β could stand for need the same storage representation — *boxing*.

- Many approaches using universal/existential types try to reduce or minimize boxing, e.g., Peyton Jones and Launchbury [1991]; Leroy [1992]. General themes of unboxing are to:
  - Inline and simplify to remove as much type polymorphism as possible.
  - Represent every member of any single type the same way.

Beyond Unboxing with Intersection/Union Types (1)

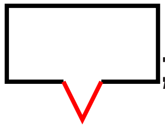
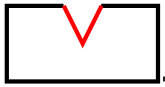
- In general, conventions are major causes of inefficiency. Examples include conventions on storage (boxing), closure layout, register usage in function calls.

- Having more information about the program lessens the need for following conventions. Universal/existential types obstruct the availability of this information while intersection/union types support it.

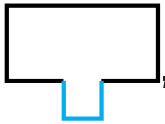
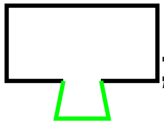
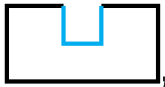
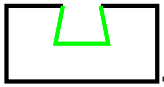
- Intersection and union types provide a structure that can guide transformations to enable avoiding conventions [Dimock, Muller, Turbak, and Wells, 1997]. This idea is illustrated on the following pages.

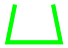
Some Abstract Notation

In the following pages, I will use some abstract notation to describe the issue of using multiple, possibly conflicting data representations.

- Pre-transformation source representation: ; sink representation: .

  For example, these may be function definitions and call sites.

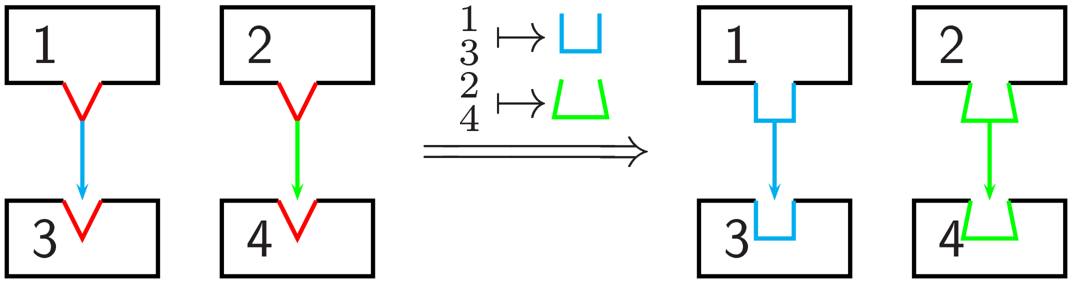
- Post-transformation source representations: , ; sink representations: , .

  For example,  may represent standard code/environment closures and  may represent code-only "closures" [Wand and Steckler, 1994].

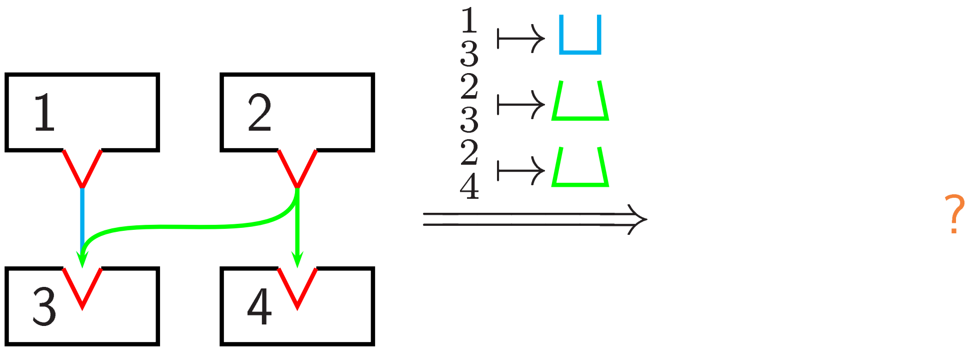
Supporting Conflicting Representations (1)

Given a flow analysis and representation choices, how to implement them?

Easy case:

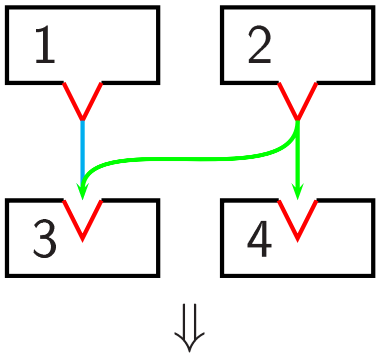


Hard case:

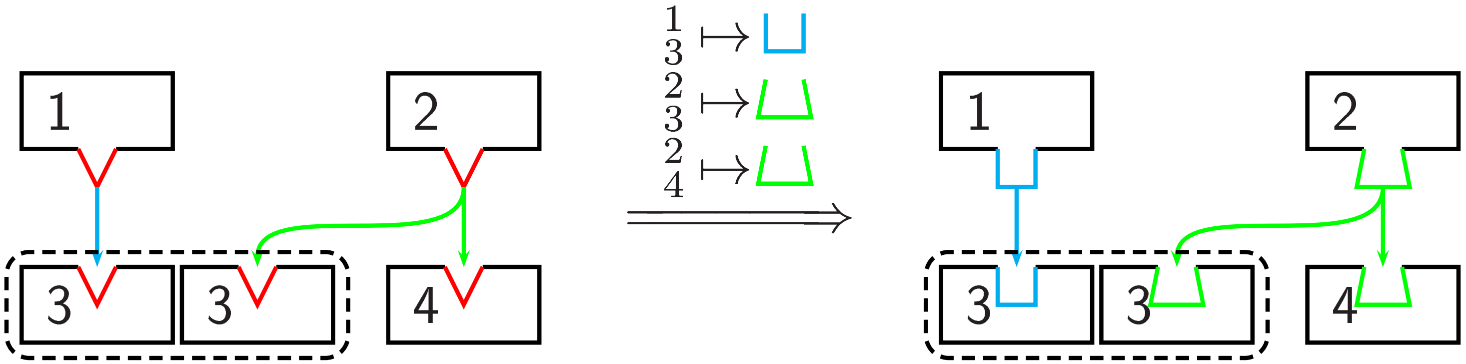


Supporting Conflicting Representations (2)

The solution uses intersection/union types to do this:



If $\sqcup$ and $\sqcup$ behavior incompatible, then real copies, else virtual copies.



Beyond Unboxing with Intersection/Union Types (2)

- Although code may be duplicated, it is reasonable to expect that subsequent inlining and simplification will counteract this.

- Values of intersection (union) type are seen as *virtual* records (variants) which can be made real if desired.

- Multiple, incompatible representations (e.g., conflicting function call representations) can be supported automatically when heuristics indicate it may be more efficient.

- We call this approach *flow/type-directed compilation*.

Overview

- The Appeal of Types.
- Type Polymorphism: What and Why.
- The Advantages of Intersection & Union Types.
  - Modularity via Principality.
  - Type and Flow Analysis
  - Compiling without Conventions (Type/Flow-Directed Compilation)
- Future Work and Conclusion.

Future Work

- Continuing development of prototype compiler using intersection/union types.
- Libraries of advanced intermediate representations. Link-time optimization and code generation.
- Engineering solutions to unnecessary redundancies in typings using intersection/union types.
- Extending the formal treatment to handle the imperative features (assigments, exceptions) which are present in the implementation.
- Proving that the transformations preserve meaning.

Conclusion

- Type polymorphism via intersection/union types is a new paradigm that has significant advantages over universal/existential types.

- Intersection types are best for program *analysis* because they have *principality*.

- Type analysis *is* flow analysis. Furthermore, intersection/union types are easier to use for flow analysis and the principality of intersection types allows modular flow analysis.

- Intersection/union types help in avoiding the inefficiencies of conventions.

References

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In POPL '99 POPL '99 [1999], pages 147–160.

Steffen van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 ICFP '97 [1997].

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

Mario Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560. Academic Press, 1980.

L. Damas and Robin Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 207–212, 1982.

Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 ICFP '97 [1997], pages 11–24.

Jean-Yves Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse d'Etat, Université de Paris VII, 1972.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, 1995.

Nevin Heintze. Control-flow analysis and type systems. In *Proc. 2nd Int'l Static Analysis Symp.*, volume 983 of *LNCS*, pages 189–206, 1995.

ICFP '97. *Proc. 1997 Int'l Conf. Functional Programming*, 1997.

Suresh Jagannathan, Stephen Weeks, and Andrew Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, volume 1302 of *LNCS*. Springer-Verlag, 1997.

Thomas Jensen. Inference of polymorphic and conditional strictness properties. In POPL '98 POPL '98 [1998].

Trevor Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, pages 196–207, 1994.

Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101, 1999.

Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In POPL '99 POPL '99 [1999], pages 161–174.

Xavier Leroy. Unboxed objects and polymorphic typing. In *Conf. Rec. 19th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 177–188, 1992.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conf. Rec. 15th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 47–57, 1988.

John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In POPL '98 POPL '98 [1998].

Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. on Prog. Langs. and Systs.*, 17(4):576–599, 1995.

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In POPL '98 POPL '98 [1998], pages 197–208. Superseded by Palsberg and Pavlopoulou [1999].

Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. A substantially revised version of Palsberg and Pavlopoulou [1998]. Available at http://www.cs.purdue.edu/homes/palsberg/paper /popl98.ps.gz., February 1999.

Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symp. on Programming*, 1996.

Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA '91, Conf. Funct. Program. Lang. Comput. Arch.*, volume 523 of *LNCS*, Cambridge, MA. U.S.A., 1991. Springer-Verlag.

Benjamin Pierce. Bounded quantification is undecidable. *Inf. & Comput.*, 112: 131–165, 1994.

POPL '98. *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

POPL '99. *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, 1999.

J. C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425, Paris, France, 1974. Springer-Verlag.

Simona Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theor. Comp. Sc.*, 59:181–209, 1988.

Simona Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comp. Sc.*, 28:151–169, 1984.

Olin Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. & Comput.*, 111(2):245–296, 1994.

TAPS '97. *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, 1997.

Peter Thiemann. A unified framework for binding-time analysis. In TAPS '97 TAPS '97 [1997], pages 742–756.

Jerzy Tiuryn and Paweł Urzyczyn. The subtyping problem for second-order types is undecidable. Tech. rep., Univ. of Warsaw, November 1995.

Jerzy Tiuryn and Paweł Urzyczyn. The subtyping problem for second-order types is undecidable. In *Proc. 11th Ann. IEEE Symp. Logic in Computer Sci.*, 1996. Shorter proceedings version of Tiuryn and Urzyczyn [1995].

Franklyn Turbak, Allyn Dimock, Robert Muller, and J. B. Wells. Compiling with polymorphic and polyvariant flow types. In *First Int'l Workshop on Types in Compilation*, June 1997. Boston Coll. Comp. Sci. Dept. Tech. Rep. BCCS-97-03.

Paweł Urzyczyn. Type reconstruction in $\mathbf{F}_\omega$. *Math. Struc. in Comp. Sc.*, 7(4): 329–358, 1997.

Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, pages 435–445, 1994.

J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. Logic in Computer Sci.*, pages 176–185, 1994. Superseded by Wells [1999].

J. B. Wells. The undecidability of Mitchell's subtyping relation. Tech. Rep. 95-019, Comp. Sci. Dept., Boston Univ., December 1995.

J. B. Wells. Typability is undecidable for F+eta. Tech. Rep. 96-022, Comp. Sci. Dept., Boston Univ., March 1996a.

J. B. Wells. *Type Inference for System F with and without the Eta Rule*. Ph.D. thesis, Boston Univ., 1996b.

J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure & Appl. Logic*, 98(1–3):111–156, 1999. Supersedes Wells [1994].

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In TAPS '97 TAPS '97 [1997], pages 757–771. Superseded by Wells et al. [199X].

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Prog.*, 199X. Accepted subject to revisions. Supersedes Wells et al. [1997].