Flow-Based Function Representation Transformation

Boston Coll.	R. Muller
Boston Univ.	S. Pericas, I. Westmacott
Harvard Univ.	A. Dimock, G. Holloway
Heriot-Watt Univ.	J. Wells
Wellesley Coll.	F. Turbak

The Church Project

December 23, 1998

Overview

- The Church Project: people, goals, achievements.
- Representation Transformation Framework.
  - Overview: general ideas, related work.
  - Design: phases of algorithm, examples.
  - Status: proving correctness, implementation.

The Church Project

- Boston-based project, in memory of Alonzo Church, investigating intersection types in compilation of higher-order strongly typed languages.
- Members:

Members:	*Boston Coll.*	Bob Muller
	Boston Univ.	Assaf Kfoury, Torben Amtoft, Elena Machkasova, Santiago Pericas, Ian Westmacott
	Harvard Univ.	Allyn Dimock, Glenn Holloway
	Heriot-Watt Univ.	Joe Wells
	Stevens Inst. of Tech.	Anindya Banerjee
	Wellesley Coll.	Lyn Turbak

  Ex-students: Trevor Jim, Alex Sorg, Dan Bachiochi, Gary McDonald.
- Web address: `http://www.cs.bu.edu/groups/church`.

The Church Project: Goals

The Church Project uses intersection and union types in:

- Analyzing programs. So far this includes type and flow analysis.
- Optimizing/refining programs.

The challenges include:

- How to do it at all?
- How to gain benefits?

Both are addressed in this talk.

Intersection/Union Types

- Intersection types: finitary polymorphism for values. Example:

$$(\lambda x.x) : (\mathrm{int} \to \mathrm{int}) \wedge (\mathrm{bool} \to \mathrm{bool})$$

  Traditionally would have type $\forall\alpha.(\alpha \to \alpha)$. The example intersection type is somewhat like $\forall\alpha \in \{\mathrm{int}, \mathrm{bool}\}.(\alpha \to \alpha)$.

- Union types: dual of intersection types, finitary polymorphism for contexts/continuations. Example:

$$((\lambda x.\mathbf{not}\, x), \mathbf{true}) : ((\mathrm{bool} \to \mathrm{bool}) \times \mathrm{bool}) \vee ((\mathrm{int} \to \mathrm{bool}) \times \mathrm{int})$$

  Traditionally would have type $\exists\alpha.(\alpha \to \mathrm{bool}) \times \alpha$. The example union type is somewhat like $\exists\alpha \in \{\mathrm{int}, \mathrm{bool}\}.(\alpha \to \mathrm{bool}) \times \alpha$.

Types and Flow Analysis

- Flow labels encode flow analysis in typing [Heintze, 1995]. Example:

$$(\lambda^1 x.x@_2(\lambda^3 y.y)) : (((\sigma \xrightarrow[\psi']{3} \sigma) \xrightarrow[2]{\phi} \tau) \xrightarrow[\psi]{1} \tau)$$

- Adding intersection/union types provides *polyvariant* flow analysis [Banerjee, 1997]. Example:

$$(\lambda^1(f,g,h).(f@_2 g, f@_3 h)) : (((\sigma \xrightarrow[2]{4} \sigma) \wedge (\tau \xrightarrow[3]{4} \tau)) \times \sigma \times \tau) \xrightarrow[\psi]{1} (\sigma \times \tau)$$

where $\sigma = \mathrm{int} \xrightarrow[6]{5} \mathrm{int}$ and $\tau = \mathrm{int} \xrightarrow[8]{7} \mathrm{int}$

Somewhat like *abstract interpretation*.

The Church Project: Progress

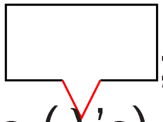
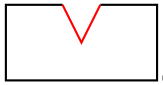
- λ^{CIL}, an explicitly typed calculus with intersection/union types [Wells et al., 1997].
- A polyvariant/polymorphic type/flow analysis algorithm using rank-2 intersection types [Banerjee, 1997].
- Framework for representation transformation [Dimock, Muller, Turbak, and Wells, 1997]. (This talk!)
- In progress: An ML compiler using the above.
- Principal typing algorithm for the full system of intersection types [Kfoury and Wells, 1999].

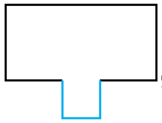
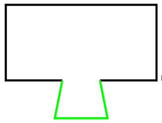
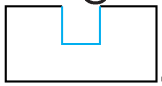
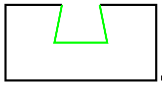
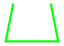
Representation Transformation Framework: Overview

- Target representations can be chosen on a per-source/sink-pair basis.
- A program's flow paths are automatically replumbed, duplicating sources or sinks as necessary, to allow incompatible:
  - Types of transformed representations.
  - Behavior of transformed representations.
- The framework has been applied to functions. This supports representations such as standard, selective, and lightweight closure conversion as well as (flow-directed) inlining.
- The framework can be used for other types, e.g., products and sums.

Some Abstract Notation

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

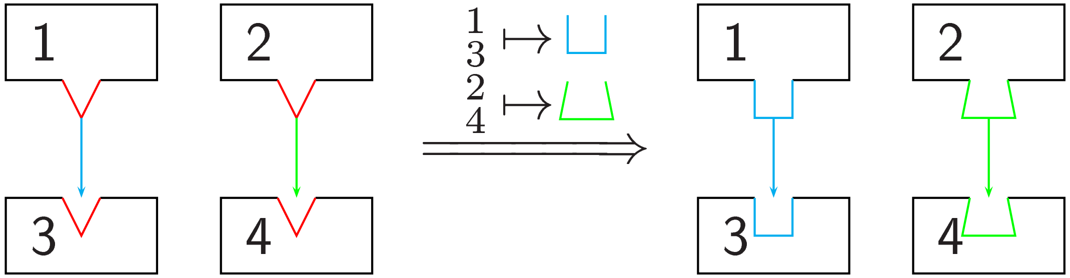
Original source representation: ; original sink representation: . For example, these may be functions (λ's) and call sites.

Target source representations: , . Target sink representations: , . For example,  may represent standard code/environment closures and  may represent code-only "closures".

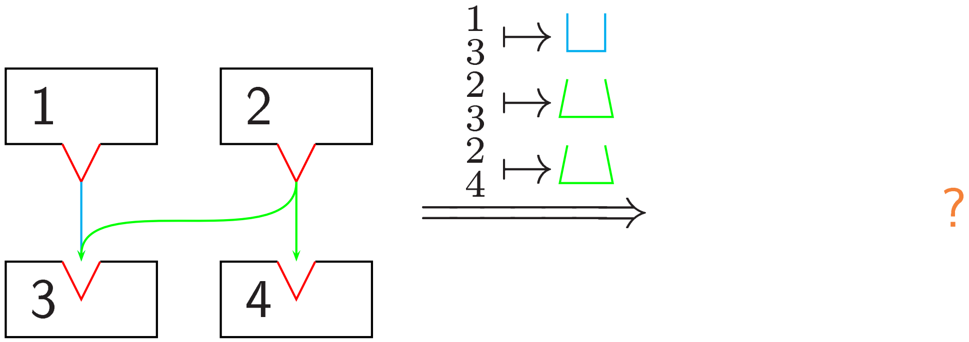
Supporting Conflicting Representations

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

Easy case:

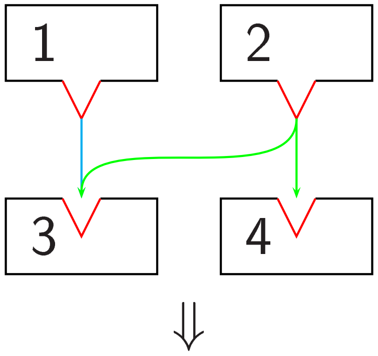


Hard case:

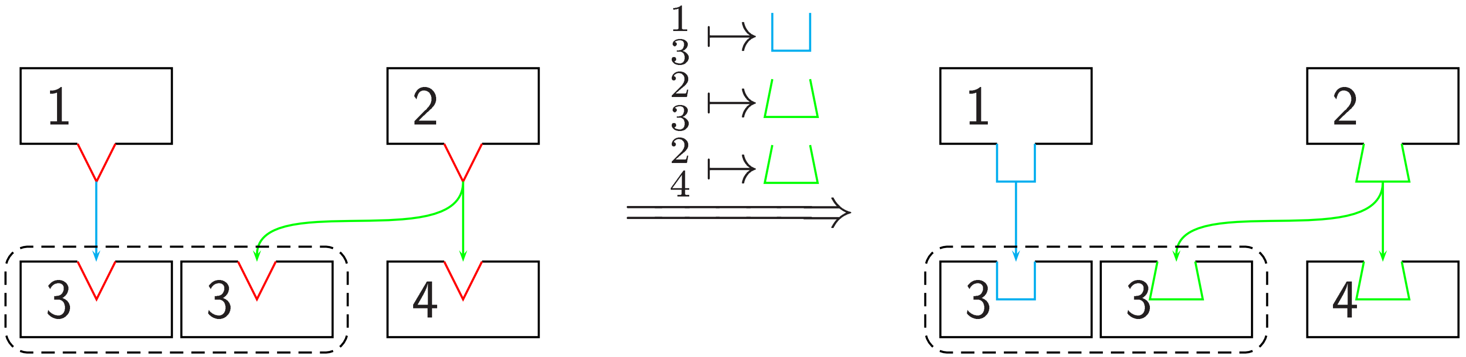


Supporting Conflicting Representations

The solution uses intersection/union types to do this:



If and behavior incompatible, then real copies, else virtual copies.



Applying the Framework to Functions

- *Closure conversion* explicitly packages free variable values with function code, allowing further IL transformations. Difficult on typed programs.
- *Inlining* moves function code to its call site, reducing call costs and allowing further optimizations.
- *Selective* closure conversion [Wand and Steckler, 1994] avoids packaging empty environments with closed functions.
- *Lightweight* closure conversion (also Wand & Steckler) avoids putting values which are available at the call site into a closure.

Our framework allows more inlining and selective/lightweight closure conversion than previous approaches.

Typed Closure Conversion

Classic example of difficulty [Minamide et al., 1996]:

$$\textbf{if } b \textbf{ then } (\lambda x^{\mathrm{int}}.y^{\mathrm{int}}) \textbf{ else } (\lambda z^{\mathrm{int}}.0) \quad : \quad \mathrm{int} \to \mathrm{int}$$

$$\Downarrow$$

$$\begin{array}{rl} \textbf{if } b \textbf{ then} & ((\lambda(x^{\mathrm{int}}, e^{(\mathrm{int})}).\pi_1 e), (y^{\mathrm{int}})) \\ \textbf{else} & ((\lambda(z^{\mathrm{int}}, e^{()}).0), (())) \end{array} \quad : \quad ?$$

Research Status:

	type constructors used	poly?	selective?	lightweight?	linked?	recursive?
Hannan [1995]	$\times$	no	yes	yes	no	no
Minamide et al. [1996]	$\exists$, $\otimes$, $\forall\alpha = t$	yes	no	no	yes	no
Tolmach [1997]	$+$, $\times$, μ	no	no	no	no	yes
our work	$\vee$, $\wedge$, $+$, $\times$, μ	yes	yes	yes	no	yes

Flow-Directed Inlining

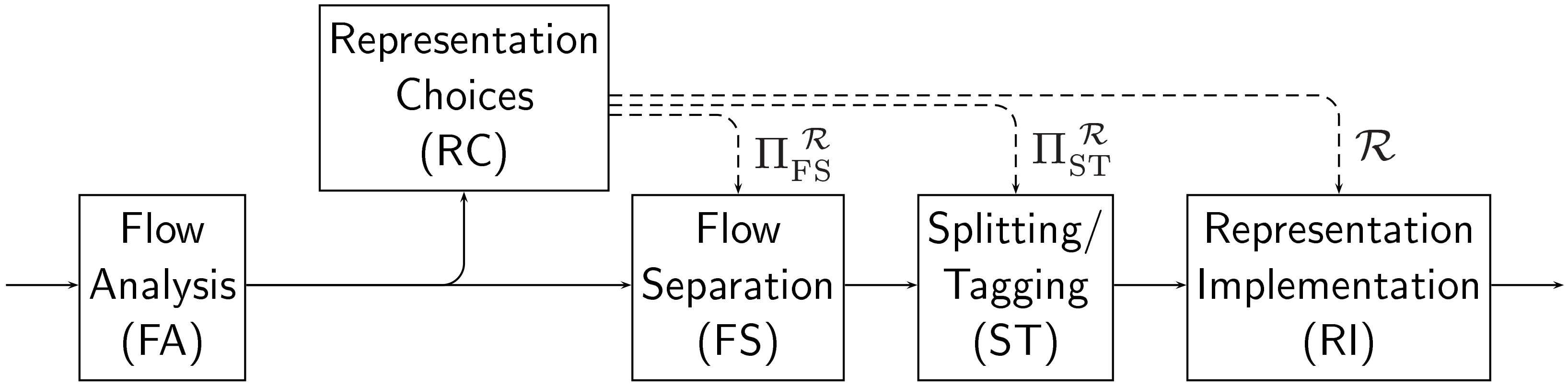
- Inlining traditionally based on syntactic cues.

- Benefits: (1) eliminating function call overhead, (2) enabling further optimizations.

- Jagannathan and Wright [1996] did flow-directed inlining for Scheme: untyped, uses let-based polyvariant flow analysis, can inline open functions, only inlines when one function flows to call site.

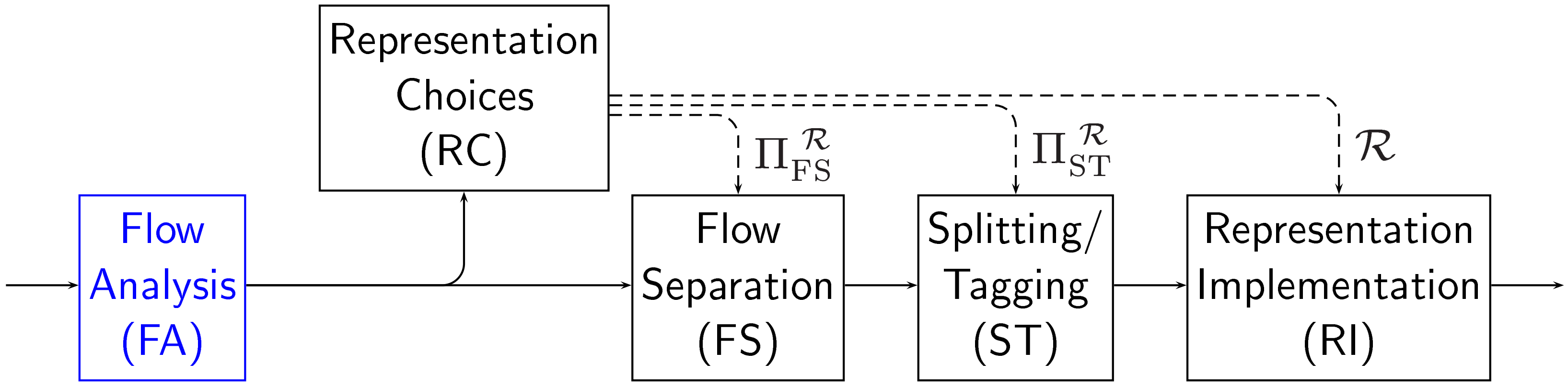
- Our framework: strongly typed, uses any flow analysis encodable in λ^{CIL}, can inline open functions, can inline when many functions flow to call site (automatically generates necessary tagging and tag inspection).

Representation Transformation Framework: Design



- FA: Flow analysis meeting minimum requirements.
- RC: Chooses representation for each flow path.
- FS: Separates flow paths at type level (fine-grained).
- ST: Separates flow paths at term level (coarser).
- RI: Installs implementations of new representations.

Phase: Flow Analysis



- Each source (sink) label must identify a producer (consumer) subterm.
- Each source/sink label pair must identify a type constructor and its component types. (Only shallow subtyping allowed.)
- Otherwise, no requirements. Can be monovariant or polyvariant. The framework doesn't care.

Example Flow Analysis

Diagrammatic display of a flow analysis encoded in λ^{CIL}:

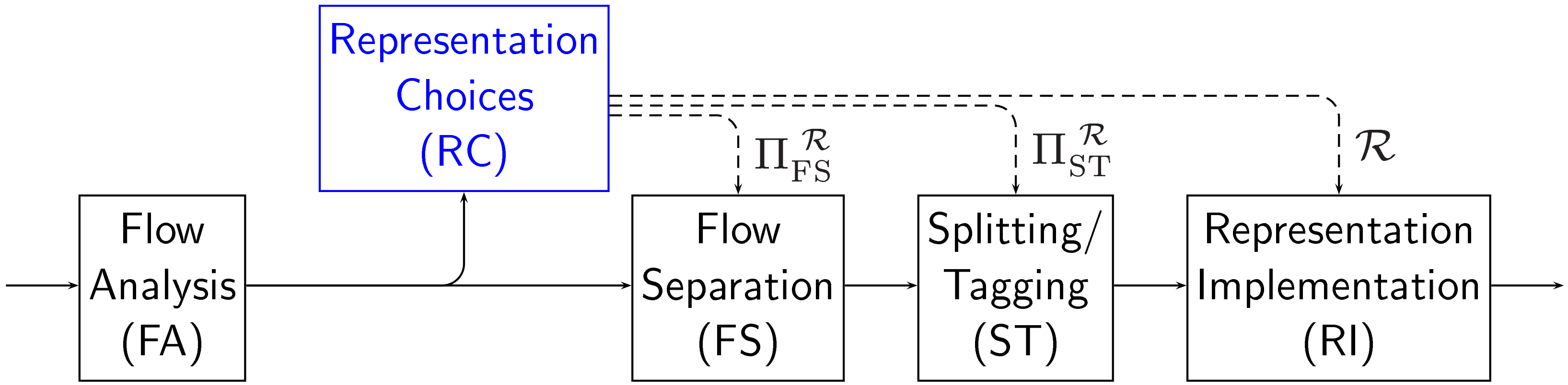
$$\boxed{\lambda^{1}_{\{3,4\}} x^{\mathrm{int}}.x * 2} \qquad \boxed{\lambda^{2}_{\{4\}} y^{\mathrm{int}}.y + a^{\mathrm{int}}}$$

$$\boxed{\square \mathbin{@}^{\{1\}}_{3} 5} \qquad \boxed{(\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ \square\ \mathbf{else}\ \square) \mathbin{@}^{\{1,2\}}_{4} 7}$$

The fragments above might have come from this whole program:

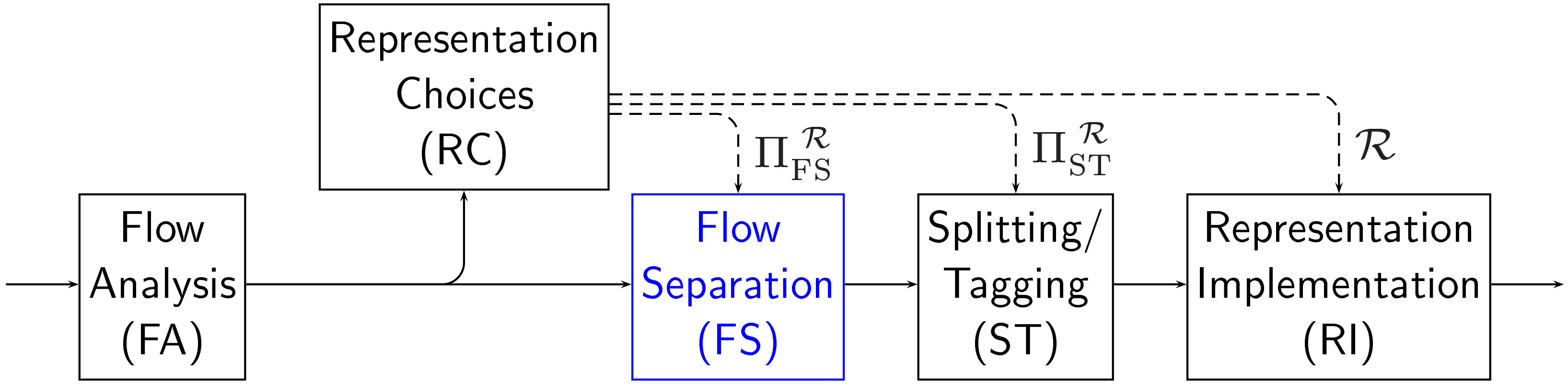
$$\begin{array}{l}
\mathbf{let}\ f^{\mathrm{int}\to\mathrm{int}} = \lambda x^{\mathrm{int}}.x * 2 \\
\mathbf{in\ let}\ g^{\mathrm{int}\to\mathrm{int}} = \lambda y^{\mathrm{int}}.y + a^{\mathrm{int}} \\
\quad \mathbf{in}\ \times\big(f \mathbin{@} 5, (\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ f\ \mathbf{else}\ g) \mathbin{@} 7\big)
\end{array}$$

Phase: Representation Choices



- Chooses a representation for each flow path (source/sink label pair).
- Generates partitions for FS and ST phases which tell where type and/or term incompatibilities have been specified and duplication is required.
- Ensures representation contraints are not violated:
  - Values of free variables of functions must reach call sites.
  - Inlining loops must not occur.

Phase: Flow Separation



- Maps types of form $\xrightarrow[\psi]{\phi}$ to form $\bigvee[\cdots, \bigwedge[\cdots, \xrightarrow[\psi_j]{\phi_i}, \cdots], \cdots]$.
- Splits sources (sinks) with incompatible representation choices into *virtual tuples* (*virtual case expressions*).
- Transforms some *flow label coercions* into unpacking/repacking of virtual tuples or retagging of virtual variants.
- Minor β-expansion for virtual case expressions (not completely virtual).

Example Flow Separation

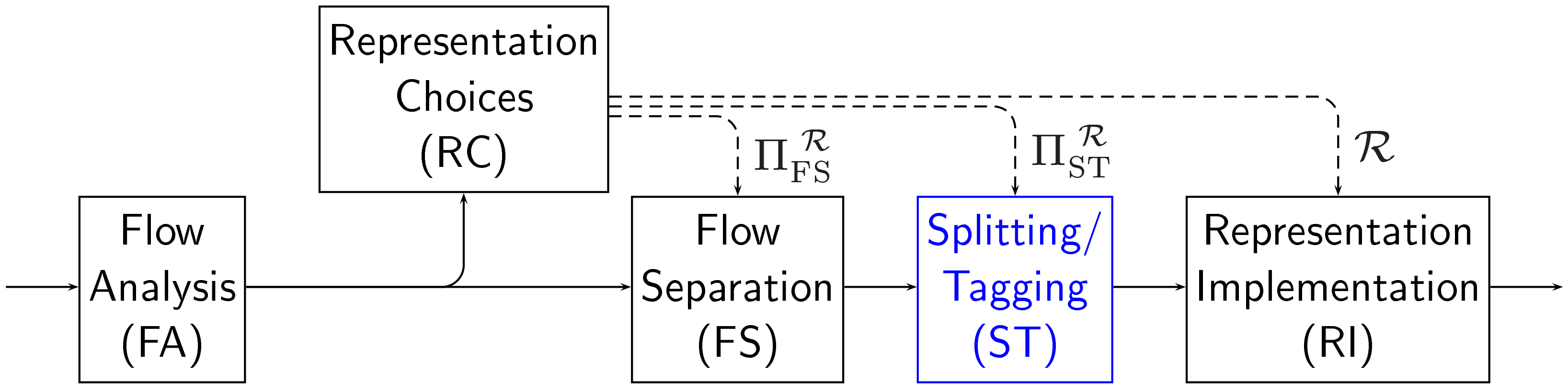
Specifying the partitioning $\Pi^{\mathcal{R}}_{\mathrm{FS}} = \{{}^1_3 \mapsto \sqcup, {}^1_4 \mapsto \sqcup\}$ yields these transformations:

$$\boxed{\lambda^1_{\{3,4\}} x^{\mathrm{int}}.x * 2} \quad \Longrightarrow \quad \boxed{\bigwedge\Big(\lambda^1_{\{3\}} x^{\mathrm{int}}.x * 2,\ \lambda^1_{\{4\}} x^{\mathrm{int}}.x * 2\Big)}$$

$$\boxed{f^{\mathrm{int} \xrightarrow[\{3,4\}]{\{1\}} \mathrm{int}} \ @^{\{1\}}_3\ 5} \quad \Longrightarrow \quad \boxed{(\pi^{\wedge}_1\ \Box)\ @^{\{1\}}_3\ 5}$$

(Subtyping coercions not shown.)

Phase: Splitting/Tagging

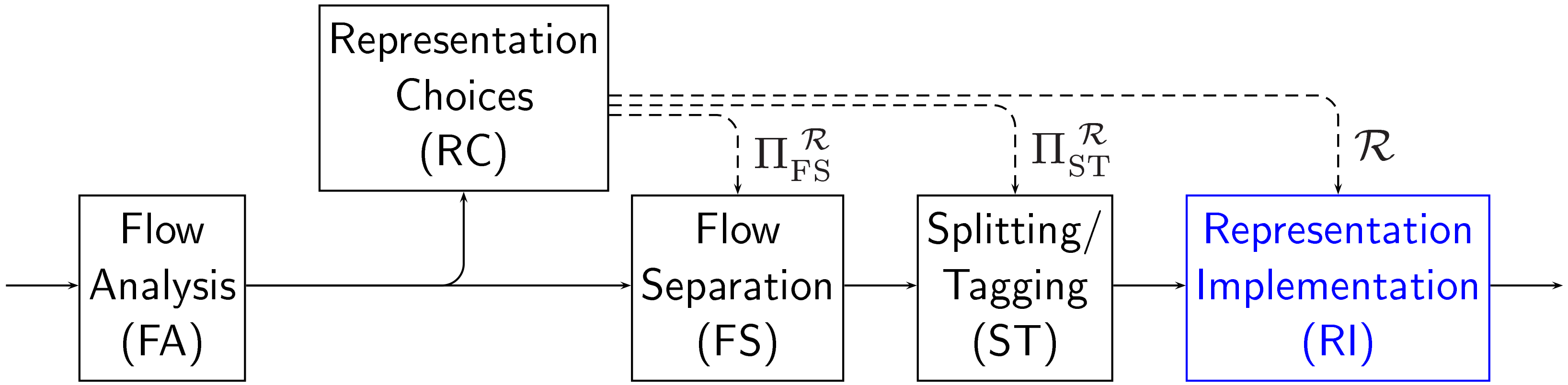


- Makes virtual tuples ($\wedge$-type) into real tuples ($\times$-type) when slots can not be shared. Similarly for virtual case expressions ($\vee$ to $+$).
- Must sometimes split outer virtual tuples, e.g.:

$$\mathbf{let}\ x = \bigwedge(\mathbf{let}\ y = \bigwedge(M_1, M_2)\ \mathbf{in}\ M_3, \mathbf{let}\ y = M_4\ \mathbf{in}\ M_5)\ \mathbf{in}\ M_6$$

Making the inner $\wedge$ into a $\times$ forces the outer $\wedge$ to change also.

Phase: Representation Implementation



- Intricately connected to RC phase. Changes in one generally necessitate changes in the other.
- Replaces sources (functions) with new representation.
- Replaces sinks (call sites) similarly; this may inline a function body.
- Replaces variables with code to retrieve value from appropriate place.

Example Transformation 1

Selective closure conversion, multiple representations for a function:

$$\boxed{\begin{aligned}&\times\Big((\lambda^{1}_{\{3\}} x^{\mathrm{int}}.x * 2),\\ &\qquad \times\Big((\lambda^{0}_{\{0\}} x^{\times[\times[\,],\mathrm{int}]}.(\pi^{\times}_{2}\ x) * 2), \times(\,)\Big)\Big)\end{aligned}}$$

$$\boxed{\begin{aligned}&\times\Big(\lambda^{0}_{\{0\}} y^{\times[\times[\mathrm{int}],\mathrm{int}]}.(\pi^{\times}_{2}\ y) + (\pi^{\times}_{1}\ \pi^{\times}_{1}\ y),\\ &\qquad \times\big(a^{\mathrm{int}}\big)\Big)\end{aligned}}$$

$$\boxed{(\pi^{\times}_{1}\ \square)\ @^{\{1\}}_{3}\ 5}$$

$$\boxed{\begin{aligned}&\mathbf{case}^{\vee}\Big(\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ \ (\mathbf{in}^{\vee}_{1}\ (\pi^{\times}_{2}\ \square))^{\vee[\sigma_1,\sigma_2]}\\ &\qquad\qquad\qquad\ \mathbf{else}\ \ (\mathbf{in}^{\vee}_{2}\ \square)^{\vee[\sigma_1,\sigma_2]}\Big)\\ &\mathbf{bind}\ h\ \mathbf{as}\ \sigma_1 \Rightarrow (\pi^{\times}_{1}\ h)\ @^{\{0\}}_{0}\ \times\big((\pi^{\times}_{2}\ h), 7\big),\\ &\qquad\qquad\quad \sigma_2 \Rightarrow (\pi^{\times}_{1}\ h)\ @^{\{0\}}_{0}\ \times\big((\pi^{\times}_{2}\ h), 7\big)\end{aligned}}$$

where $\sigma_1 = \times\Big[(\times[\times[\,], \mathrm{int}] \xrightarrow[\{0\}]{\{0\}} \mathrm{int}), \times[\,]\Big]$ and $\sigma_2 = \times\Big[(\times[\times[\mathrm{int}], \mathrm{int}] \xrightarrow[\{0\}]{\{0\}} \mathrm{int}), \times[\mathrm{int}]\Big]$

Example Transformation 2

Selective closure conversion, multiple function-call protocols at a call site:

$$\boxed{\lambda^{1}_{\{3,4\}} x^{\mathrm{int}}.x * 2}$$

$$\boxed{\times\Big(\lambda^{0}_{\{0\}} y^{\times[\times[\mathrm{int}],\mathrm{int}]}.(\pi^{\times}_{2}\; y) + (\pi^{\times}_{1}\; \pi^{\times}_{1}\; y),\; \times\big(a^{\mathrm{int}}\big)\Big)}$$

$$\boxed{\square \;@^{\{1\}}_{3}\; 5}$$

$$\boxed{\begin{array}{l}\mathbf{case}^{+}\left(\begin{array}{l}\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ \left(\mathbf{in}^{+}_{1}\ \square\right)^{+\left[\mathrm{int}\xrightarrow[\{4\}]{\{1\}}\mathrm{int},\,\sigma_2\right]} \\ \qquad\qquad\ \ \mathbf{else}\ \left(\mathbf{in}^{+}_{2}\ \square\right)^{+\left[\mathrm{int}\xrightarrow[\{4\}]{\{1\}}\mathrm{int},\,\sigma_2\right]}\end{array}\right) \\ \mathbf{bind}\ h\ \mathbf{as}\ \left(\mathrm{int}\xrightarrow[\{4\}]{\{1\}}\mathrm{int}\right) \Rightarrow h\ @^{\{1\}}_{4}\ 7, \\ \qquad\qquad \sigma_2 \Rightarrow (\pi^{\times}_{1}\ h)\ @^{\{0\}}_{0}\ \times\big((\pi^{\times}_{2}\ h), 7\big)\end{array}}$$

Example Transformation 3

Selective closure conversion, inlining, multiple function-call protocols at a call site:

$$\boxed{\lambda^{1}_{\{3,4\}} x^{\mathrm{int}}.x * 2}$$

$$\boxed{\times\left(a^{\mathrm{int}}\right)}$$

$$\boxed{\Box\ @^{\{1\}}_{3}\ 5}$$

$$\boxed{\begin{array}{l}\mathbf{case}^{+}\left(\begin{array}{l}\mathbf{if}\ b^{\mathrm{bool}}\ \mathbf{then}\ \left(\mathbf{in}^{+}_{1}\ \Box\right)^{+\left[\mathrm{int}\ \xrightarrow[\{4\}]{\{1\}}\ \mathrm{int},\ \times[\mathrm{int}]\right]}\\ \qquad\qquad\ \mathbf{else}\ \left(\mathbf{in}^{+}_{2}\ \Box\right)^{+\left[\mathrm{int}\ \xrightarrow[\{4\}]{\{1\}}\ \mathrm{int},\ \times[\mathrm{int}]\right]}\end{array}\right)\\ \mathbf{bind}\ h\ \mathbf{as}\ (\mathrm{int}\ \xrightarrow[\{4\}]{\{1\}}\ \mathrm{int}) \Rightarrow h\ @^{\{1\}}_{4}\ 7,\\ \qquad\qquad \times[\mathrm{int}] \Rightarrow 7 + (\pi^{\times}_{1}\ h)\end{array}}$$

Representation Transformation Framework: Status

- Theoretical status: what's been proven correct?
- Implementation status: what's running now?
- Future work: what's needed next?

Meaning Preservation

- Meaning preservation can only be expected for closed programs whose type does not mention $\rightarrow$, $\wedge$, or $\vee$.
- Proven for FS and ST phases for λ^{CIL} (without state or exceptions).
- Not yet proven for full framework.
- We *think* the specification is correct.

Implementation Progress

- Standard ML compiler is in progress. Uses front-end parser, type checker, and elaborator from SML/NJ compiler and MLRISC code generator also from SML/NJ.
- The RC, FS, ST, and RI phases are running for some cases, but are not yet fully debugged.
- The FA phase uses a modified version of Banerjee [1997]. Not yet completely working on full language (i.e., recursive bindings, state, exceptions).
- The front and back ends seem to be working.

Future Work

- Extend λ^{CIL} to account for extensions in the implementation: named products/sums, flow labels on all types, multi-arg functions, state, exceptions, etc.
- λ^{CIL} extensions: record/variant subtyping, case rule ignoring untriggered clauses, deep subtyping, register/stack allocation, etc.
- Evaluation of various heuristics for representation choices.
- Evaluation of various flow analyses.
- Standard optimizations to take advantage of transformation.
- Engineering of efficient representation of λ^{CIL}.

Related Work Not Already Mentioned

- Intersection/union types: van Bakel [1993], Jim [1996], Pierce [1991], Aiken, Wimmers, and Lakshman [1994], Eifrig, Smith, and Trifonov [1995], Barbanera, Dezani-Ciancaglini, and de'Liguoro [1995].

- Flow-directed inlining: Plevyak and Chien [1995].

Conclusion

- Intersection/union types can be used in the compiler intermediate language for a function-oriented language.
- Intersection/union flow types enable a representation transformation framework which supports far greater flexibility in inlining and efficient closure representations.
- There is a lot of work to be done — there are many opportunities for others to join and contribute. I will be happy to tell you more if you ask.

References

Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In POPL '94 POPL [1994], pages 163–173.

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 [1997].

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

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

Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.* Elsevier, 1995.

John Hannan. Type systems for closure conversion. In *Workshop on Types for Program Analysis*, pages 48–62, 1995. DAIMI PB-493.

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

Proc. 1997 Int'l Conf. Functional Programming, 1997.

Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proc. ACM SIGPLAN '96 Conf. Prog. Language Design & Implementation*, pages 193–205, 1996.

Trevor Jim. What are principal typings and what are they good for? In POPL '96 POPL [1996].

Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Principles of Prog. Languages*, 1999.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In POPL '96 POPL [1996].

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

John Plevyak and Andrew Chien. Type directed cloning for object-oriented programs. In *Workshop for Languages and Compilers for Parallel Computers*, 1995.

Conf. Rec. 21st Ann. ACM Symp. Principles of Programming Languages, 1994.

Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages, 1996.

Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *First Int'l Workshop on Types in Compilation*, 1997. Boston Coll. Comp. Sci. Dept. Tech. Rep. BCCS-97-03.

Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In POPL '94 POPL [1994], pages 435–445.

J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, 1997. 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. To appear. Supersedes Wells et al. [1997].