Semantics of Program Analyses and Transformations

Lecture Notes for the Summer School on Program Analysis and Transformation Copenhagen, Denmark 6-10 June 2005

Nick Benton
Microsoft Research
`nick@microsoft.com`

1 Introduction

These lectures are concerned with how we can formalize just what it is that a static analysis computes about programs and how we might begin to formalize the ways in which that information may be used in program transformation.

The most obvious reason for trying to formalize what a static analysis computes is to be able to prove that it is *sound*: whenever the analysis reports that a program P satisfies a property ϕ ($\vdash P : \phi$), then it really *does* do so ($\models P : \phi$).[1] The approach we take is to this problem is semantic: we will define the meaning $[\![\phi]\!]$ of each property ϕ in a way which is independent of the analysis system or algorithm that is used to assign properties to programs. The semantic approach should be contrasted with the common (at least in the case of type systems) syntactic one. The syntactic approach, pioneered by Wright and Felleisen [54], manages to avoid ever giving an independent interpretation to program properties, relying instead on a form of subject reduction (preservation and progress) to show that derivable properties behave well with respect to small-step reductions in an operational semantics.

Where should we look for the meanings $[\![\phi]\!]$ of properties? For simple properties,[2] $[\![\phi]\!]$ will be a set. We can either work directly with the syntax, taking the interpretation of a property to be a set of programs, which is natural if we're

[1] It is worth remarking that there are useful but unsound analyses. Bug-finding tools (such as PREfix) perform static analyses but may be unsound in the sense that they generate both false positives and false negatives. One would not wish to use such an analysis as the basis of automated transformations in a compiler, but these tools only produce a report indicating *potential* bugs – it is the programmer's responsibility to decide whether, and if so how, to transform the program. We will not (intentionally, at least) discuss unsound analyses here.

[2] We will see that not all properties are simple in this sense.

going to define $\models$ using an operational semantics, or go via a denotational semantics in some mathematical space D, taking the interpretation of a property to be a set of denotations (i.e. $[\![P]\!] \in D$, $[\![\phi]\!] \subseteq D$, and we then define $\models P : \phi$ to mean $[\![P]\!] \in [\![\phi]\!]$).

Whether we base our reasoning on operational or denotational techniques, however, there is a further important choice to be made: how extensional our definition of $[\![\phi]\!]$ should be. This is intimately connected with the use to which we intend to put the information gathered by our analysis, and to our desire to separate the meaning of properties from any specific syntactic system or algorithm for inferring those properties of programs.

Analysis systems (at least, the implementable ones) are generally said to compute safe approximations, since interesting properties of programs are uncomputable. In particular, the computed results of static analyses are not generally closed under contextual equivalence: we can have $\vdash P : \phi$ and $P \sim P'$, but $\not\vdash P' : \phi$. In other words, the behaviour of the program analysis algorithm depends on intensional, non-observable (from within the language) details of the source program, and this is essentially unavoidable if the analysis is computable. But there is generally a whole range of possible "degrees of extensionality" for the interpretations of properties. If our intent is to use the results of the analysis to decide whether we can perform some transformation $f : \mathit{programs} \to \mathit{programs}$, then a plausible-looking[3] chain of implications for establishing correctness is

$$\vdash P : \phi \quad \Longrightarrow \quad \models P : \phi \quad \Longrightarrow \quad P \sim f(P)$$

Intuitively, we have flexibility in choosing the relative 'lengths' of the two arrows. At one extreme, we could choose such a syntactic and intensional interpretation of properties that $\models P : \phi \iff \vdash P : \phi$ holds (the analysis could then even be claimed to be complete). At the other end of the scale, we could try to define the properties in terms of the transformations we wish to perform: $\models P : \phi \iff P \sim f(P)$.[4]

One possible objection to either of these extremes is that they break the problem down into rather unequal pieces: the interposition of the middle step seems not to simplify the overall problem much. Another is that both $\vdash$ and f are very specific, referring to the syntax of a particular language, and also to a particular analysis or transformation. Since there are many analyses that can justify a particular transformation, and, indeed, many transformations that can be justified by a given analysis, we'd like to define $\models P : \phi$ in a way which is maximally reusable as well as intellectually satisfying.[5]

[3]We shall see that this is actually a bit too naive.

[4]'This work suggests that the proposition associated with a program analysis can simply be that "the optimization works." ' [51].

[5]This point seems embarassingly subjective and non-scientific, but I'll try to make it anyway. Proofs about programming languages rarely establish theorems the truth or falsity of which has direct practical significance. We do them to gain sufficient insight and understanding to find, fix and avoid bugs in real artifacts about which we cannot reason fully formally, and to be able to design better artifacts, such as more efficient, accurate, expressive or useful

Correctness proofs for analyses, especially for imperative programs, have traditionally tended to use interpretations of properties which lie towards the intensional end of the scale. Typically, one defines an *instrumented* semantics that tracks extra information about how computations execute (e.g. which variables have been read or written), beyond that which would appear in a standard semantics for the language. Whether the instrumented semantics is presented operationally (e.g. [38, 48], amongst many others) or denotationally (e.g. [34]), there is something slightly unsatisfactory about this approach (which is closely related to the syntactic approach to type soundness). Of course, some analyses are clearly intended to compute non-extensional properties (e.g. relating to time or space usage) and the interpretations of those properties must be relative to a semantics that makes those aspects of computation explicit.[6] But for analyses which are going to be used to justify the actual legality of a transformation, it seems that sufficient preconditions for the transformation should be expressible in terms of a standard extensional semantics for the language on which we perform the transformation (which may be a compiler intermediate language). If the source and the target are indeed observationally equivalent, then we should be able to explain *why* without messing about with the semantics we first thought of.

As a facetious example of the difference between the intensional and extensional approaches, consider why the following transformation is correct:

```
X := 7;                X := 7;
Y := Y+1;      ==>     Y := Y+1;
Z := X;                Z := 7;
```

The extensional answer is "when `X` is evaluated on the last line, its value will always be 7". The intensional answer is something like "the only definition of `X` which reaches its use on line 3 is the one on line 1, and the right hand side of that definition does not contain any variable which is assigned to in lines 1 or 2". This may well be an accurate account of *how* an algorithm works, but it is not a good basis for thinking about *what* it establishes. Things get even worse if we consider a sequence like

```
X := 7;           X := 7;           X := 7;
Y := Y+1;   ==>   Y := Y+1;   ==>   Y := Y+1;
X := 7;           X := 7;
Z := X;           Z := 7;           Z := 7;
```

After the first transformation, the intensional justification for the change to line 4 refers to the definition of `X` on line 3. But after the second transformation, that definition has gone, which complicates proving the correctness of the combined transformation. Problems of this sort occur both in real compilers (keeping

analyses and transformations. The shortest path to 'QED' may not be the most informative, or useful in suggesting a better theorem.

[6]Importantly, this includes analyses or heuristics for *profitability*, used to decide whether there is likely to be a benefit in performing some legal transformation.

analysis results sound during transformations is notoriously tricky) and in proofs (see for example the discussion of interference between 'forward' analyses and 'backward' transformation patterns in [31]).

There are a number of possible advantages from avoiding instrumentation in formulating the soundness of analyses:

1. Different analyses are all interpreted relative to the same semantics rather than each requiring their own ad hoc extension.

2. We're in a much better position to reason about transformations. An instrumented semantics, because it tracks extra intensional information, will have a weaker equational theory than the source language.

3. We gain much more independence from the arbitrary syntactic details of analysis systems. The same extensional interpretation of properties should work for analyses that differ in precision.

4. We gain a deeper understanding.

But neither will we here go to the other extreme of just defining $\models P : \phi$ directly in terms of transformations.[7] Instead, we'll try to define the meanings of analyses extensionally, in terms either of a standard non-instrumented computationally adequate denotatational semantics (or, operationally, using sets of programs that are closed under observational equivalence). I believe this position on our intensionality scale seems to be maximally informative and maximally reusable, leading to interpretations of program properties that are useful both for many analyses and many applications.

We will abstract away from algorithmic details of analyses by presenting them declaratively, using type-like inference systems. We will also ignore the difficult question of how one decides, or specifies formally, exactly *how* one wishes transform a program after analysis: we merely give inference systems that define a *set* of programs, each of which is equivalent to the original one given the results of the analysis.

Our basic approach is applicable to many different sorts of analysis, and for different styles of programming language. In Section 2 we consider the soundness of strictness analysis for a pure, higher-order, call-by-name language. In Section 3 we consider the semantics of classical dataflow analyses for imperative programs, and the soundness of the transformations they enable. Finally, in Section 4, we give a denotational semantics to a simple effect analysis for an impure, higher-order, call-by-value language and show how it enables program transformations.

[7]Though this is actually rather a promising idea. A nice paper by Führmann [19] suggests that, removing the dependency on a particular complex transformation f by instead looking at allowable patterns of local rewrites (e.g. which computations can be duplicated, discarded or commuted in which contexts), it *may* be possible to develop an elegant and general algebraic approach to the meanings of analyses.

2 Strictness Analysis

The problem of how to justify transforming call-by-name (CBN) into call-by-value (CBV) has been studied by (arguably far too) many researchers, starting with Mycroft's work in 1980 [35] right up to the present [44]. It's not been a hot research topic for a decade or so, but still provides a nice case study. The material in this section is based on [16, 15], closely related work may be found in [26, 25].

Languages such as Haskell have a CBN semantics. In particular, this means that the operational behaviour of function application is specified (using big-step style) by a rule like

$$\frac{M[N/x] \Downarrow V}{(\lambda x.M)\, N \Downarrow V}$$

Understood naively, this looks to be expensive, since it makes multiple copies of N, many of which may be evaluated during the evaluation of $M[N/x]$. In the absence of side-effects in the language, however, each of these evaluations will yield equivalent results, so there is a uniformly applicable (i.e. not dependent on any static analysis) optimization one can perform (at a lower level than the source language), called lazy or call-by-need parameter passing, that implements function calls by passing arguments as self-updating thunks. If the evaluation of M requires the value of x, the thunk for N is evaluated, which not only computes a value V', but also overwrites itself with that value, so subsequent evaluations of x will yield V' immediately. As the replacement of call-by-name with call-by-need is not based on a program analysis, we will not discuss it further here[8] except to note that it is still more expensive than CBV evaluation. Creating thunks to delay the evaluation of arguments has a cost, as does the code for forcing them (which may involve an indirection or a test to check if the thunk has already been evaluated) and the extra work involved in doing the updates. But just using CBV evaluation:

$$\frac{N \Downarrow V' \qquad M[V'/x] \Downarrow V}{(\lambda x.M)\, N \Downarrow V}$$

in place of CBN will change the observable behaviour of programs. If the evaluation of M does not require the value of x (for example, if there are no free occurrences of x in M) then N will not be evaluated at all under CBN, but will be evaluated under CBV. So if Ω is a divergent term, then a program such as $(\lambda x.3)\,\Omega$ will diverge under CBV but converge under CBN.

So we would like to selectively replace CBN with CBV, i.e. to compile some applications (and possibly function definitions) to use CBV and some to use CBN. When is this safe? If one thinks intensionally, then the above discussion

[8]Though its correctness is still non-trivial to show, and it gives a clear demonstration that the shape of derivations in a natural operational semantics one might use for specifying a language can bear a decidedly non-trivial relation to execution steps in a real implementation, which is a good reason not to take them too seriously when formalizing analyses if one wishes the theorems to apply to actual implementations.

leads one towards the answer "when one can detect that the evaluation of the function body will always require its argument". Several researchers (e.g. [55]) have developed *neededness* analyses, which can detect this property, and even related it to relevance logics [7]. But neededness is a very intensional property, the formalization of which involves a detailed syntactic analysis of the fine structure of reduction (labelling subterms and tracking those labels through the operational semantics, which is a form of instrumentation). Furthermore, neededness is (at least, if formalized naively) unduly conservative as a precondition for our transformation. For example, the function

$$\lambda b.\lambda n.\texttt{if}\ b\ \texttt{then}\ n+1\ \texttt{else}\ \Omega$$

does not need (always evaluate) its second argument, n, yet it *is* always safe to pass that argument by value: even if the argument diverges and the function wouldn't have tried to evaluate it, the function body would have diverged anyway. This observation leads us to reformulate the precondition for the transformation as "when one can detect that the function diverges whenever the argument diverges". This property, *strictness*, is strictly weaker than neededness (so is true of more programs) and, crucially, is extensional: it is defined entirely in terms of the observable input/output behaviour of functions. Thus, even if we had an analysis that only detected neededness, giving it a semantics in terms of strictness is likely to be neater and more tractable. Of course, once we've realised that strictness is the more general property, we would probably rather have a more powerful analysis that looks directly for strictness.

We now give a more detailed account of a strictness analysis and how we may formulate its meaning and correctness using a standard denotational semantics.

2.1 The Λ Language

Our language, Λ, is a simply-typed CBN functional programming language, like Plotkin's PCF [39]. This is all entirely standard, but the next two sections briefly recall the syntax, type rules, operational semantics and denotational semantics of Λ to fix notation.

The types of the language are

$$A, B \ ::= \ \texttt{int} \mid \texttt{bool} \mid A \to B$$

the terms are

$$\begin{array}{rcl} M, N & ::= & x \mid \underline{n} \mid \underline{b} \mid M + N \mid M = N \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid \\ & & \lambda x : A.M \mid M\,N \mid \texttt{rec}(x : A, M) \end{array}$$

where x ranges over some set of variables, b and n range over booleans and integers, respectively, $+$ and $=$ are representative arithmetic and comparison operators. The type rules are standard, and shown in Figure 1.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \underline{n} : \texttt{int}} \qquad \frac{}{\Gamma \vdash \underline{b} : \texttt{bool}}$$

$$\frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M + N : \texttt{int}} \qquad \frac{\Gamma \vdash M : \texttt{int} \quad \Gamma \vdash N : \texttt{int}}{\Gamma \vdash M = N : \texttt{bool}}$$

$$\frac{\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N_1 : A \quad \Gamma \vdash N_2 : A}{\Gamma \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \ : A} \qquad \frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \texttt{rec}(x : A, M) : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{M\, N : B}$$

Figure 1: Types for Λ

$$\frac{}{V \Downarrow V}$$

$$\frac{M \Downarrow \underline{m} \quad N \Downarrow \underline{n}}{M + N \Downarrow \underline{m + n}} \qquad \frac{M \Downarrow \underline{m} \quad N \Downarrow \underline{n}}{M = N \Downarrow \underline{m = n}}$$

$$\frac{M \Downarrow \underline{\texttt{true}} \quad N_1 \Downarrow V}{\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \Downarrow V} \qquad \frac{M \Downarrow \underline{\texttt{false}} \quad N_2 \Downarrow V}{\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \Downarrow V}$$

$$\frac{M \Downarrow \lambda x : A.M' \quad M'[N/x] \Downarrow V}{M\, N \Downarrow V} \qquad \frac{M[\texttt{rec}(x : A, M)/x] \Downarrow V}{\texttt{rec}(x : A, M) \Downarrow V}$$

Figure 2: Operational semantics of Λ

2.2 Semantics

We define the values, V, to be a subset of terms:

$$V \ ::= \ \underline{n} \mid \underline{b} \mid \lambda x : A.M$$

and define the operational semantics of Λ by a big-step natural semantics relating closed (having no free variables) terms to closed values as shown in Figure 2. We define a context $C[\cdot] \in (\Gamma, A)^\top$ to be a 'term with a hole in', such that whenever $\Gamma \vdash M : A$, $C[M]$ is a closed term of ground (i.e. `int` or `bool`) type. We define *contextual equivalence*, $\sim$, by

$$\begin{array}{rcl}
\Gamma \vdash M_1 \sim M_2 : A & \iff & \Gamma \vdash M_1 : A \\
 & \wedge & \Gamma \vdash M_2 : A \\
 & \wedge & \forall C[\cdot] \in (\Gamma, A)^\top . C[M_1] \Downarrow \iff C[M_2] \Downarrow
\end{array}$$

Note that we only observe termination at ground type.

A denotational semantics for Λ may be given in the category of ω-cpos and continuous maps. Recall that the objects D of this category are partially ordered sets $(|D|, \sqsubseteq)$ that have least upper bounds $\bigsqcup d_i$ for all ω-indexed ascending chains $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$ The morphisms are functions between the underlying sets that are monotonic (preserve the order relation) and preserve least upper bounds of chains. The category is cartesian closed: products are given by the product of the underlying sets with the pointwise order, whilst the exponential $A \Rightarrow B$ is the set of continuous functions $A \to B$ with the order inherited from the codomain. We say D is pointed if it has a least element $\perp_D$. If D is pointed then any morphism $f : D \to D$ has a least fixed point, $fix(f) \in D$, given by $fix(f) = \bigsqcup_{i \in \omega} f^i(\perp_D)$. There is a lift monad $(\cdot)_\perp$ which adds a least element to a cpo: $|D_\perp|$ is $\{\lceil d \rceil | d \in D\} \cup \{\perp\}$ and the order relation is the obvious one from D extended with $\perp \sqsubseteq \lceil d \rceil$ for all $d \in D$.

The interpretation $[\![A]\!]$ of each type A as a cpo is given inductively:

$$\begin{array}{rcl} [\![\texttt{int}]\!] & = & \mathbb{Z}_\perp \\ [\![\texttt{bool}]\!] & = & \mathbb{B}_\perp \\ [\![A \to B]\!] & = & [\![A]\!] \Rightarrow [\![B]\!] \end{array}$$

and if $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then $[\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$. (Note that these cpos are all pointed.)

If $\Gamma \vdash M : A$ then we define $[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to [\![A]\!]$ by induction as shown in Figure 3. We should really state and prove a few lemmas about weaking, substitution, etc. and verify that the semantics we've presented really is well-defined, but we omit those standard details here.

The important result about the relationship between the operational and denotational semantics is the following, which is established using a logical relation between syntax and semantics [15, 53]:

Theorem 1 (Adequacy). *If* $[\![\Gamma \vdash M_1 : A]\!] = [\![\Gamma \vdash M_2 : A]\!]$ *then* $\Gamma \vdash M_1 \sim M_2 : A$.

In other words, equality of denotations implies contextual equivalence. The converse, full abstraction, does not hold for this semantics because of the presence of non-sequential functions in the model [39].

2.3 Strictness Logic

We will define a strictness analysis by giving a logic that refines the existing type system of Λ. For each type A, we define a set of propositions $\mathcal{L}_A$ as follows:

$$\frac{}{t^A \in \mathcal{L}_A} \qquad \frac{}{f^A \in \mathcal{L}_A}$$

$$\frac{\phi \in \mathcal{L}_A \quad \psi \in \mathcal{L}_B}{\phi \to \psi \in \mathcal{L}_{A \to B}} \qquad \frac{\phi \in \mathcal{L}_A \quad \psi \in \mathcal{L}_A}{\phi \wedge \psi \in \mathcal{L}_A}$$

$$[\![x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i]\!]\,\rho \;=\; \pi_i(\rho)$$

$$[\![\Gamma \vdash \underline{n} : \texttt{int}]\!]\,\rho \;=\; \lceil n \rceil$$

$$[\![\Gamma \vdash \underline{b} : \texttt{bool}]\!]\,\rho \;=\; \lceil b \rceil$$

$$[\![\Gamma \vdash M + N : \texttt{int}]\!]\,\rho \;=\; \begin{cases} \lceil m + n \rceil & \text{if } [\![\Gamma \vdash M : \texttt{int}]\!]\,\rho = \lceil m \rceil \\ & \text{and } [\![\Gamma \vdash N : \texttt{int}]\!]\,\rho = \lceil n \rceil \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\Gamma \vdash M = N : \texttt{bool}]\!]\,\rho \;=\; \begin{cases} \lceil m = n \rceil & \text{if } [\![\Gamma \vdash M : \texttt{int}]\!]\,\rho = \lceil m \rceil \\ & \text{and } [\![\Gamma \vdash N : \texttt{int}]\!]\,\rho = \lceil n \rceil \\ \bot & \text{otherwise} \end{cases}$$

$$\begin{aligned} &[\![\Gamma \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 : A]\!]\,\rho \\ &\quad = \begin{cases} [\![\Gamma \vdash N_1 : A]\!]\,\rho & \text{if } [\![\Gamma \vdash M : \texttt{bool}]\!]\,\rho = \lceil \texttt{true} \rceil \\ [\![\Gamma \vdash N_2 : A]\!]\,\rho & \text{if } [\![\Gamma \vdash M : \texttt{bool}]\!]\,\rho = \lceil \texttt{false} \rceil \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

$$[\![\Gamma \vdash \lambda x : A.M : A \to B]\!]\,\rho \;=\; \lambda d \in [\![A]\!].[\![\Gamma, x : A \vdash M : B]\!]\,(\rho, d)$$

$$[\![\Gamma \vdash \texttt{rec}(x : A, M) : A]\!]\rho \;=\; \mathit{fix}(\lambda d \in [\![A]\!].\,[\![\Gamma, x : A \vdash M : A]\!]\,(\rho, d))$$

Figure 3: Denotational Semantics of Λ

and we define an entailment preorder $\leq_A \subseteq \mathcal{L}_A \times \mathcal{L}_A$ as shown in Figure 4. We now define a refined type X to be a pair (A, ϕ) where $\phi \in \mathcal{L}_A$ and a refined context Θ to be a finite map from variables to refined types. We will abbreviate (A, ϕ) to ϕ^A, and if $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ then Θ^Γ ranges over refined contexts of the form $x_1 : (A_1, \phi_1), \ldots, x_n : (A_n, \phi_n)$. Our program logic, or refined type system, for strictness analysis is shown in Figure 5.

The intuition is that the property f^A will abstract divergent terms of type A, whereas t represents all terms of type A ('don't know'). Thus, for example, given some $\vdash M : \texttt{int} \to \texttt{int}$, if we can derive $\vdash M : f^{\texttt{int}} \to f^{\texttt{int}}$ in the strictness logic, we will be able to deduce that M is strict. We will make this intended interpretation precise by giving a semantics to strictness properties as subsets of the cpos used in the denotational semantics of Λ. We will then prove soundness by showing that if the judgement that a term has a certain property is derivable, then the denotation of that term is indeed an element of the the denotation of the property.

$$\frac{\phi \in \mathcal{L}_A}{f^A \leq_A \phi} \qquad \frac{\phi \in \mathcal{L}_A}{\phi \leq_A t^A} \qquad \frac{\phi \in \mathcal{L}_A}{\phi \leq_A \phi}$$

$$\frac{\phi \in \mathcal{L}_A \quad \psi \in \mathcal{L}_A}{\phi \wedge \psi \leq_A \phi} \qquad \frac{\phi \in \mathcal{L}_A \quad \psi \in \mathcal{L}_A}{\phi \wedge \psi \leq_A \psi}$$

$$\frac{\phi \leq_A \psi \quad \phi \leq_A \chi}{\phi \leq_A \chi} \qquad \frac{\phi \leq_A \psi_1 \quad \phi \leq_A \psi_2}{\phi \leq_A \psi_1 \wedge \psi_2}$$

$$\frac{\phi' \leq_A \phi \quad \psi \leq_B \psi'}{(\phi \to \psi) \leq_{A \to B} (\phi' \to \psi')}$$

$$\frac{}{t^{A \to B} \leq_{A \to B} (t^A \to t^B)} \qquad \frac{}{t^A \to f^B \leq_{A \to B} f^{A \to B}}$$

$$\frac{\phi \in \mathcal{L}_A \quad \psi_1 \in \mathcal{L}_B \quad \psi_2 \in \mathcal{L}_B}{(\phi \to \psi_1) \wedge (\phi \to \psi_2) \leq_{A \to B} (\phi \to \psi_1 \wedge \psi_2)}$$

Figure 4: Strictness Logic Entailment

We define the semantics of extended types as follows:

$$\begin{array}{rcl}
[\![(A,\phi)]\!] & \subseteq & [\![A]\!] \\
[\![(A,t^A)]\!] & = & [\![A]\!] \\
[\![(A,f^A)]\!] & = & \{\bot_{[\![A]\!]}\} \\
[\![(A,\phi \wedge \psi) & = & [\![(A,\phi)]\!] \cap [\![(A,\psi)]\!] \\
[\![(A \to B, \phi \to \psi)]\!] & = & \{f \in [\![A \to B]\!] \mid \forall a \in [\![(A,\phi)]\!].\, f(a) \in [\![(B,\psi)]\!]\}
\end{array}$$

and then define $[\![\Theta^\Gamma]\!] \subseteq [\![\Gamma]\!]$ in the natural pointwise manner:

$$[\![x_1 : (A_1,\phi_1), \ldots x_n : (A_n,\phi_n)]\!] \;=\; [\![(A_1,\phi_1)]\!] \times \cdots \times [\![(A_n,\phi_n)]\!]$$

Lemma 1. *For any well formed extended type (A,ϕ), $[\![(A,\phi)]\!]$ is an* ideal *(a non-empty, Scott-closed subset) of $[\![A]\!]$. That is:*

1. *$\bot_{[\![A]\!]} \in [\![(A,\phi)]\!]$.*
2. *$[\![(A,\phi)]\!]$ is down-closed. If $a \in [\![(A,\phi)]\!]$ and $a' \sqsubseteq a$ then $a' \in [\![(A,\phi)]\!]$.*
3. *$[\![(A,\phi)]\!]$ is closed under limits of chains. If $a_0 \sqsubseteq a_1 \sqsubseteq \ldots$ is a chain in $[\![A]\!]$ such that $a_i \in [\![(A,\phi)]\!]$ for all i, then $\bigsqcup_i a_i \in [\![(A,\phi)]\!]$.*

A simple induction, using entirely set-theoretic reasoning, established the soundness of the entailment rules shown in Figure 4:

Lemma 2. *If $\phi \leq_A \psi$ is derivable then $[\![(A,\phi)]\!] \subseteq [\![(A,\psi)]\!]$.*

$$\frac{}{\Theta, x : \phi^A \vdash x : \phi^A} \qquad \frac{}{\Theta \vdash \underline{n} : t^{\texttt{int}}} \qquad \frac{}{\Theta \vdash \underline{b} : t^{\texttt{bool}}}$$

$$\frac{\Theta \vdash M : f^{\texttt{int}} \quad \Theta \vdash N : \phi^{\texttt{int}}}{\Theta \vdash M + N : f^{\texttt{int}}} \qquad \frac{\Theta \vdash M : \phi^{\texttt{int}} \quad \Theta \vdash N : f^{\texttt{int}}}{\Theta \vdash M + N : f^{\texttt{int}}}$$

$$\frac{\Theta \vdash M : f^{\texttt{int}} \quad \Theta \vdash N : \phi^{\texttt{int}}}{\Theta \vdash M = N : f^{\texttt{bool}}} \qquad \frac{\Theta \vdash M : \phi^{\texttt{int}} \quad \Theta \vdash N : f^{\texttt{int}}}{\Theta \vdash M = N : f^{\texttt{bool}}}$$

$$\frac{\Theta \vdash M : t^{\texttt{int}} \quad \Theta \vdash N : t^{\texttt{int}}}{\Theta \vdash M + N : t^{\texttt{int}}} \qquad \frac{\Theta \vdash M : t^{\texttt{int}} \quad \Theta \vdash N : t^{\texttt{int}}}{\Theta \vdash M = N : t^{\texttt{bool}}}$$

$$\frac{\Theta \vdash M : f^{\texttt{bool}} \quad \Theta \vdash N_1 : t^A \quad \Theta \vdash N_2 : t^A}{\Theta \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 : f^A}$$

$$\frac{\Theta \vdash M : t^{\texttt{bool}} \quad \Theta \vdash N_1 : \phi^A \quad \Theta \vdash N_2 : \phi^A}{\Theta \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 : \phi^A}$$

$$\frac{\Theta, x : \phi^A \vdash M : \phi^A}{\Theta \vdash \texttt{rec}(x : A, M) : \phi^A}$$

$$\frac{\Theta, x : \phi^A \vdash M : \psi^B}{\Theta \vdash (\lambda x : A.M) : (\phi \to \psi)^{A \to B}} \qquad \frac{\Theta \vdash M : (\phi \to \psi)^{A \to B} \quad \Theta \vdash N : \phi^A}{\Theta \vdash M\, N : \psi^B}$$

$$\frac{\Theta \vdash M : \phi^A \quad \phi \leq_A \psi}{\Theta \vdash M : \psi^A} \qquad \frac{\Theta \vdash M : \phi^A \quad \Theta \vdash M : \psi^A}{\Theta \vdash M : (\phi \wedge \psi)^A}$$

Figure 5: Strictness Logic

and, although we don't need this for soundness, we note that our axiomatisation of entailment is actually complete:

Lemma 3. *If $\phi, \psi \in \mathcal{L}_A$ and $[\![(A, \phi)]\!] \subseteq [\![(A, \psi)]\!]$ then $\phi \leq_A \psi$.*

Soundness of the program logic is then an induction on the rules in Figure 5:

Theorem 2. *If $\Theta^\Gamma \vdash M : (A, \phi)$ and $\rho \in [\![\Theta^\Gamma]\!]$ then $[\![\Gamma \vdash M : A]\!]\,\rho \ \in\ [\![(A, \phi)]\!]$.*

The proof of Theorem 2 makes use of Lemma 1 to establish the correctness of the rule for recursion (ideals are *admissible* subsets).

So the strictness logic correctly lets us derive strictness-related facts about the denotations of terms. For example:

- If $\vdash M : f^{\texttt{int}} \to f^{\texttt{int}}$ then $[\![M]\!]$ is strict.
- If $\vdash M : t^{\texttt{int}} \to f^{\texttt{int}}$ then $[\![M]\!]$ is the constant $\perp$ function.

- If $\vdash M : t^{\mathtt{int}} \to f^{\mathtt{int}} \to f^{\mathtt{int}}$ then $[\![M]\!]$ is a curried function that is strict in its second argument.
- If $\vdash M : f^{\mathtt{int}} \to f^{\mathtt{int}} \to f^{\mathtt{int}}$ then $[\![M]\!]$ is *jointly strict* in its two arguments.
- If $\vdash M : (t^{\mathtt{int}} \to f^{\mathtt{int}} \to f^{\mathtt{int}}) \wedge (f^{\mathtt{int}} \to t^{\mathtt{int}} \to f^{\mathtt{int}})$ then $[\![M]\!]$ is a curried function that is strict in *both* its arguments.
- If $\vdash M : (f^{\mathtt{int}} \to f^{\mathtt{int}}) \to (f^{\mathtt{int}} \to f^{\mathtt{int}})$ then $[\![M]\!]$ is a higher-order function mapping strict functions to strict functions.
- If $\vdash M : (f^{\mathtt{int}} \to f^{\mathtt{int}}) \to f^{\mathtt{int}}$ then $[\![M]\!]$ is a higher order function that returns $\bot$ when given a strict argument.
- etc.

The reader may enjoy seeing what Theorems 1 and 2 imply about the soundness of the logic in operational terms – it turns out that we have actually generalized the naive operational notion of strictness we had at the start of the section.

Unfortunately, there still seems to be no entirely satisfactory formalization of, let alone correctness proof for, the ways in which one may use the results of this kind of higher-order strictness analysis in program transformation, though simpler neededness-based transformations have been shown sound [5, 8], and there is at least one other serious attempt to explain the use of strictness information in compilation [37].

3 Analyses and Transformations for Simple Imperative Programs

The fact that strictness-based transformations haven't been nicely formalized and proved correct is slightly embarrassing. In fact, there has been comparatively little serious work on justifying any analysis-based program transformations, even though many analyses have been proved correct in isolation. That's because it's a hard problem. Much of the work which *has* been done on proving transformations is from the group at Northeastern [51, 46, 50, 49] led by Mitch Wand, who says, for example [51],

> The goal of (flow) analysis is to annotate a program with certain propositions about the behavior of that program. One can then apply optimizations to the program that are justified by those propositions. However, it has proven remarkably difficult to specify the semantics of those propositions in a way that justifies the resulting optimizations.

Similar observations have been made by many other researchers.[9]

[9] I believe this has bad effects in practice. It's not so much that real optimizing compilers perform unsound transformations (though that is certainly a real problem), but that the use they make of expensively-gathered information is rather conservative.

In this section, based on [9, 11], we will formalize and show the correctness of some very simple analyses *and* associated transformations for imperative programs. One interesting feature is that these kinds of analyses are usually presented in a very intensional way indeed, for example

> An assignment $[x := a]^l$ *may reach* a certain program point if there is an execution of the program where x was last assigned a value at l when the program point is reached.

Nevertheless, we will show that it is possible to express *what* it is that such analyses compute, as opposed to *how* they compute it, in terms of an extensional semantics.

The second notable feature is that we move beyond a simple interpretation of properties as sets, to a semantics in which properties are binary relations. Typed lambda calculi are routinely presented using judgements of the form

$$\Gamma \vdash M = M' : A$$

which does not assert "under assumptions Γ, M equals M' and they both have type A", but rather "under assumptions Γ, M and M' are equal *at* type A". Such calculi can be modelled by interpreting types as partial equivalence relations over some untyped universe such as D_∞. Many program analyses are presented as non-standard type systems, and partial equivalence relations have been used to give semantics to these non-standard types (equivalently, elements of abstract domains), at least in the cases of binding-time [24] and security analyses [43]. However, even in those cases, the emphasis has been on simple typing judgements rather than equational reasoning. Our approach is to treat all abstract properties as relations, including those which have naive interpretations as predicates (e.g. 'X is 5'), and to present transformations by giving rules for deriving (non-standard) typed equations in context.

3.1 The Language of `while`-Programs

The syntax and denotational semantics of the language of `while`-programs are entirely standard (see, e.g. [53]). To fix notation, they are briefly summarized in Figure 6. We sometimes use F_τ as a metavariable ranging over τ `exp` where $\tau \in \{\texttt{int}, \texttt{bool}\}$.

The denotational semantics is given in the same category of ω-complete partial orders that we used for Λ. We write $(\cdot)^* : (D \to D'_\perp) \to (D_\perp \to D'_\perp)$ for the Kleisli extension operation of the lift monad. When $R \subseteq D \times D$, $R_\perp \subseteq D_\perp \times D_\perp$ is the relation defined by

$$R_\perp = \{(\lceil x \rceil, \lceil y \rceil) \mid (x, y) \in R\} \cup \{(\perp, \perp)\}$$

If $f : D \to E$, $x \in D$ and $y \in E$ then we define $f[x \mapsto y] : D \to E$ in the usual way:

$$(f[x \mapsto y])(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

Syntax

$$
\begin{array}{rcl}
X \in \mathbb{V} & = & \{\mathtt{X}, \mathtt{Y}, \ldots\} \\
n \in \mathbb{Z} & & b \in \mathbb{B} = \{true, false\} \\
iop & \in & \{+, \times, -, \ldots\} \ \subseteq \ \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
bop & \in & \{<, =, \ldots\} \ \subseteq \ \mathbb{Z} \times \mathbb{Z} \to \mathbb{B} \\
lop & \in & \{\vee, \wedge, \ldots\} \ \subseteq \ \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\
\mathtt{int\ exp} \ni E & := & \mathtt{n} \mid X \mid E \ \mathtt{iop}\ E \\
\mathtt{bool\ exp} \ni B & := & \mathtt{b} \mid E \ \mathtt{bop}\ E \mid \mathtt{not}\ B \mid B \ \mathtt{lop}\ B \\
\mathtt{com} \ni C & := & \mathtt{skip} \mid X \mathtt{:=} E \mid C \mathtt{;} C \mid \mathtt{if}\ B \ \mathtt{then}\ C \ \mathtt{else}\ C \mid \mathtt{while}\ B \ \mathtt{do}\ C
\end{array}
$$

Denotational Semantics

$$S \in \mathbb{S} = \mathbb{V} \to \mathbb{Z}$$

$$[\![E]\!] \in \mathbb{S} \to [\![\mathtt{int}]\!] = \mathbb{S} \to \mathbb{Z}$$

$$
\begin{array}{rcl}
[\![\mathtt{n}]\!]S & = & n \\
[\![X]\!]S & = & S(X) \\
[\![E_1 \ \mathtt{iop}\ E_2]\!]S & = & ([\![E_1]\!]S)\ iop\ ([\![E_2]\!]S)
\end{array}
$$

$$[\![B]\!] \in \mathbb{S} \to [\![\mathtt{bool}]\!] = \mathbb{S} \to \mathbb{B}$$

$$
\begin{array}{rcl}
[\![\mathtt{b}]\!]S & = & b \\
[\![E_1 \ \mathtt{bop}\ E_2]\!]S & = & ([\![E_1]\!]S)\ bop\ ([\![E_2]\!]S) \\
[\![B_1 \ \mathtt{lop}\ B_2]\!]S & = & ([\![B_1]\!]S)\ lop\ ([\![B_2]\!]S) \\
[\![\mathtt{not} B]\!]S & = & \neg([\![B]\!]S)
\end{array}
$$

$$[\![C]\!] \in \mathbb{S} \to \mathbb{S}_\perp$$

$$
\begin{array}{rcl}
[\![\mathtt{skip}]\!] & = & \lambda S.\lceil S \rceil \\
[\![X \mathtt{:=} E]\!] & = & \lambda S.\lceil S[X \mapsto [\![E]\!]S] \rceil \\
[\![C_1 \mathtt{;} C_2]\!] & = & [\![C_2]\!]^* \circ [\![C_1]\!] \\
[\![\mathtt{if}\ B \ \mathtt{then}\ C_1 \ \mathtt{else}\ C_2]\!] & = & \lambda S.[\![B]\!]S \implies [\![C_1]\!]S \ \mid \ [\![C_2]\!]S \\
[\![\mathtt{while}\ B \ \mathtt{do}\ C]\!] & = & fix\ f.\lambda S.[\![B]\!]S \implies f^*([\![C]\!]S) \mid \lceil S \rceil
\end{array}
$$

Figure 6: Syntax and Semantics of `while` Programs

The denotational semantics is fully abstract with respect to the obvious operational semantics and definition of observational equivalence.

If X is a set, a binary relation $R \subseteq X \times X$ is a partial equivalence relation (PER) if it is symmetric and transitive. A relation on the carrier of a pointed ω-cpo D is admissible if $(\bot, \bot) \in R$ and for all ascending chains $\langle d_i \rangle$ and $\langle d'_i \rangle$ with $(d_i, d'_i) \in R$, we have $(\sqcup_i d_i, \sqcup_i d'_i) \in R$. If R is a relation on a set X, then $R_\bot$ is an admissible relation on the flat cpo $X_\bot$ and is a PER if R is. The set of PERs on a set is closed under arbitrary intersections and disjoint unions. The set of admissible relations on a pointed cpo is closed under arbitrary intersections and finite unions. If R and S are relation on predomains D and E respectively, we write $R \Rightarrow S$ for the relation on the function space $D \to E$ defined by $(f, g) \in R \Rightarrow S$ iff $\forall (x, y) \in R.(fx, gy) \in S$. This is a PER if R and S are. If E is pointed and S is admissible, then $R \Rightarrow S$ is admissible.

3.2 Dependency, Dead Code and Constants

In this section we present DDCC, a simple analysis and transformation system for `while`-programs which tracks dependency and constancy information, enabling optimizations such as constant-folding and dead-code elimination. As indicated in the introduction, the system is presented as a non-standard type system for deriving typed equalities between expressions and between commands.

3.2.1 DDCC Syntax and Semantics

Formulae We begin by defining the syntax of some non-standard types for expressions. For $\tau \in \{\texttt{int}, \texttt{bool}\}$, $c \in [\![\tau]\!]$:

$$\phi_\tau \quad := \quad \mathbb{F}_\tau \mid \{c\}_\tau \mid \Delta_\tau \mid \mathbb{T}_\tau$$

Intuitively, $\{c\}_\tau$ is the type of τ-expressions equal to the constant c, Δ_τ is the type of τ-expressions whose value we do not know, whilst $\mathbb{T}_\tau$ is the type of τ-expressions whose value we do not care about. $\mathbb{F}_\tau$ is an empty expression type, which we have included for completeness.[10] Semantically, the denotation of ϕ_τ is a binary relation on $[\![\tau]\!]$:

$$\begin{array}{rcl} [\![\mathbb{F}_\tau]\!] & = & \emptyset \\ [\![\{c\}_\tau]\!] & = & \{(c, c)\} \\ [\![\Delta_\tau]\!] & = & \{(x, x) \mid x \in [\![\tau]\!]\} \\ [\![\mathbb{T}_\tau]\!] & = & [\![\tau]\!] \times [\![\tau]\!] \end{array}$$

Types for states are then finite maps from variables to types for `int exp`s, written as lists with the usual conventions. In particular, writing $\Phi, X : \phi_{\texttt{int}}$ implies that X does not occur in Φ.

$$\Phi \quad := \quad - \mid \Phi, X : \phi_{\texttt{int}}$$

[10]This is really just a matter of taste. $\mathbb{F}_\tau$ does not appear in many interesting derivations.

State types are interpreted as binary relations on $\mathbb{S}$:

$$\begin{array}{rcl}
[\![-]\!] & = & \mathbb{S} \times \mathbb{S} \\
[\![\Phi, X : \phi_{\mathtt{int}}]\!] & = & [\![\Phi]\!] \cap \{(S, S') \mid (S(X), S'(X)) \in [\![\phi_{\mathtt{int}}]\!]\}
\end{array}$$

Entailment There is a subtyping relation $\leq$ on expression types, which is axiomatised as follows:

$$\mathbb{F}_\tau \leq \phi_\tau \qquad \{c\}_\tau \leq \Delta_\tau$$

$$\phi_\tau \leq \mathbb{T}_\tau \qquad \phi_\tau \leq \phi_\tau$$

$$\frac{\phi_\tau \leq \phi'_\tau \quad \phi'_\tau \leq \phi''_\tau}{\phi_\tau \leq \phi''_\tau}$$

The above induces a depth- and width-subtyping relation on state types:

$$\Phi \leq - \qquad\qquad \Phi, X : \mathbb{F}_{\mathtt{int}} \leq \Phi'$$

$$\frac{\Phi \leq \Phi'}{\Phi \leq \Phi', X : \mathbb{T}_{\mathtt{int}}} \qquad \frac{\Phi \leq \Phi' \quad \phi_{\mathtt{int}} \leq \phi'_{\mathtt{int}}}{\Phi, X : \phi_{\mathtt{int}} \leq \Phi', X : \phi'_{\mathtt{int}}}$$

Because $\Phi, X : \mathbb{T}_{\mathtt{int}} \leq \Phi$ and $\Phi \leq \Phi, X : \mathbb{T}_{\mathtt{int}}$, absence of a variable from a state type is equivalent to it being present with type $\mathbb{T}_{\mathtt{int}}$.

Lemma 4.

1. *For all ϕ_τ and Φ, $[\![\phi_\tau]\!]$ and $[\![\Phi]\!]$ are partial equivalence relations.*
2. *The $\leq$ relation on state types is reflexive and transitive.*
3. *If $\phi_\tau \leq \phi'_\tau$ then $[\![\phi_\tau]\!] \subseteq [\![\phi'_\tau]\!]$.*
4. *If $\Phi \leq \Phi'$ then $[\![\Phi]\!] \subseteq [\![\Phi']\!]$.*
5. *$(S, S') \in [\![\Phi, X : \phi]\!]$ iff $\forall m, n.(S[X \mapsto m], S'[X \mapsto n]) \in [\![\Phi]\!]$ and $(S(X), S'(X)) \in [\![\phi]\!]$.*

3.2.2 Judgements

DDCC has two basic forms of judgement. For expressions, with $F, F' \in \tau\ \mathtt{exp}$, we have judgements of the form

$$\vdash F \sim F' : \Phi \Rightarrow \phi_\tau$$

whilst for commands, $C, C' \in \mathtt{com}$, there are judgements of the form

$$\vdash C \sim C' : \Phi \Rightarrow \Phi'$$

We write $\vdash C : \Phi \Rightarrow \Phi'$ as shorthand for $\vdash C \sim C : \Phi \Rightarrow \Phi'$ and similarly for single-subject expression judgements. If we define

$$\begin{array}{rcl}
[\![\Phi \Rightarrow \phi_\tau]\!] & \subseteq & (\mathbb{S} \rightarrow [\![\tau]\!]) \times (\mathbb{S} \rightarrow [\![\tau]\!]) \\
& \equiv & \{(f, f') \mid \forall (S, S') \in [\![\Phi]\!].\ (fS, f'S') \in [\![\phi_\tau]\!]\}
\end{array}$$

$$\begin{array}{rcl}
[\![\Phi \Rightarrow \Phi']\!] & \subseteq & (\mathbb{S} \rightarrow \mathbb{S}_\perp) \times (\mathbb{S} \rightarrow \mathbb{S}_\perp) \\
& \equiv & \{(f, f') \mid \forall (S, S') \in [\![\Phi]\!].\ (fS, f'S') \in [\![\Phi']\!]_\perp\}
\end{array}$$

then the intended meanings of the judgements are:

$$\begin{array}{rcl}
\models F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau & \equiv & ([\![F_\tau]\!], [\![F'_\tau]\!]) \in [\![\Phi \Rightarrow \phi_\tau]\!] \\
\models C \sim C' : \Phi \Rightarrow \Phi' & \equiv & ([\![C]\!], [\![C']\!]) \in [\![\Phi \Rightarrow \Phi']\!]
\end{array}$$

Lemma 5. *$[\![\Phi \Rightarrow \phi_\tau]\!]$ is a PER and $[\![\Phi \Rightarrow \Phi']\!]$ is an admissible PER.*

Some basic rules for deriving DDCC judgements are shown in Figure 7. The rules for expressions refer to abstract versions $\widehat{op}$ of each primitive binary operator *op* in the language. A typical definition is that for multiplication:

$\widehat{\times}$	$\mathbb{F}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\{n\}_{\texttt{int}}$	$\Delta_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$
$\mathbb{F}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$
$\{0\}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$
$\{m\}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\{m \times n\}_{\texttt{int}}$	$\Delta_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$
$\Delta_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\Delta_{\texttt{int}}$	$\Delta_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$
$\mathbb{T}_{\texttt{int}}$	$\mathbb{F}_{\texttt{int}}$	$\{0\}_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$	$\mathbb{T}_{\texttt{int}}$

The general correctness condition for abstract operations is familiar from abstract interpretation:

Definition 1. *We say $\widehat{op}$ soundly abstracts the operation op if*

$$\forall (x, x') \in [\![\phi_\tau]\!], (y, y') \in [\![\phi'_\tau]\!].\ (x\ op\ y, x'\ op\ y') \in [\![\phi_\tau\, \widehat{op}\, \phi'_\tau]\!].$$

The most interesting of the rules in Figure 7 are those for conditionals and `while`-loops. Observe that for two conditionals to be related, not only do their true and false branches have to be pairwise related, but they also have to agree on which branch is taken; this is expressed by the use of $\Delta_{\texttt{bool}}$ in the premises of the rule. Similar considerations apply to the rule for `while`-loops, which ensures that related loops execute in lockstep.

3.2.3 Equations

Using only the rules in Figure 7, most of the interesting judgements one can prove relate a phrase to itself at some type. In other words, they constitute an analysis system but not yet a program transformation system. However, the advantage of our formulation is that program transformations can now be specified and justified simply by adding new inference rules whose soundness may be straightforwardly and independently checked in terms of the semantics.

Subtyping and Structural

$$\vdash C \sim C' : \Phi, X : \mathbb{F}_{\texttt{int}} \Rightarrow \Phi' \text{ [D-CT]} \qquad \vdash F_\tau \sim F'_\tau : \Phi \Rightarrow \mathbb{T}_\tau \text{ [D-ET1]}$$

$$\vdash F_\tau \sim F'_\tau : \Phi, X : \mathbb{F}_{\texttt{int}} \Rightarrow \phi_\tau \text{ [D-ET2]} \qquad \frac{\vdash F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau}{\vdash F'_\tau \sim F_\tau : \Phi \Rightarrow \phi_\tau} \text{ [D-ESym]}$$

$$\frac{\vdash F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau \quad \Phi' \leq \Phi \quad \phi_\tau \leq \phi'_\tau}{\vdash F_\tau \sim F'_\tau : \Phi' \Rightarrow \phi'_\tau} \text{ [D-ESub]}$$

$$\frac{\vdash C \sim C' : \Phi_1 \Rightarrow \Phi_2 \quad \Phi'_1 \leq \Phi_1 \quad \Phi_2 \leq \Phi'_2}{\vdash C \sim C' : \Phi'_1 \Rightarrow \Phi'_2} \text{ [D-CSub]}$$

$$\frac{\vdash F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau \quad \vdash F'_\tau \sim F''_\tau : \Phi \Rightarrow \phi_\tau}{\vdash F_\tau \sim F''_\tau : \Phi \Rightarrow \phi_\tau} \text{ [D-ETr]}$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi'}{\vdash C' \sim C : \Phi \Rightarrow \Phi'} \text{ [D-CSym]}$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi' \quad \vdash C' \sim C'' : \Phi \Rightarrow \Phi'}{\vdash C \sim C'' : \Phi \Rightarrow \Phi'} \text{ [D-CTr]}$$

Expressions

$$\vdash X \sim X : \Phi, X : \phi_{\texttt{int}} \Rightarrow \phi_{\texttt{int}} \text{ [D-V]} \qquad \vdash \texttt{n} \sim \texttt{n} : \Phi \Rightarrow \{n\}_{\texttt{int}} \text{ [D-N]}$$

$$\vdash \texttt{b} \sim \texttt{b} : \Phi \Rightarrow \{b\}_{\texttt{bool}} \text{ [D-B]}$$

$$\frac{\vdash F_\tau \sim G_\tau : \Phi \Rightarrow \phi_\tau \quad \vdash F'_\tau \sim G'_\tau : \Phi \Rightarrow \phi'_\tau}{\vdash F_\tau \texttt{ op } F'_\tau \sim G_\tau \texttt{ op } G'_\tau : \Phi \Rightarrow (\phi_\tau \ \widehat{op} \ \phi'_\tau)} \text{ [D-}op\text{]}$$

Commands

$$\vdash \texttt{skip} \sim \texttt{skip} : \Phi \Rightarrow \Phi \text{ [D-Skip]}$$

$$\frac{\vdash C_1 \sim C'_1 : \Phi \Rightarrow \Phi' \quad \vdash C_2 \sim C'_2 : \Phi' \Rightarrow \Phi''}{\vdash (C_1 ; C_2) \sim (C'_1 ; C'_2) : \Phi \Rightarrow \Phi''} \text{ [D-Seq]}$$

$$\frac{\vdash E \sim E' : \Phi, X : \phi_{\texttt{int}} \Rightarrow \phi'_{\texttt{int}}}{\vdash X \texttt{:=} E \sim X \texttt{:=} E' : \Phi, X : \phi_{\texttt{int}} \Rightarrow \Phi, X : \phi'_{\texttt{int}}} \text{ [D-Ass]}$$

$$\frac{\vdash B \sim B' : \Phi \Rightarrow \Delta_{\texttt{bool}} \quad \vdash C \sim C' : \Phi \Rightarrow \Phi}{\vdash (\texttt{while } B \texttt{ do } C) \sim (\texttt{while } B' \texttt{ do } C') : \Phi \Rightarrow \Phi} \text{ [D-Whl]}$$

$$\frac{\vdash B \sim B' : \Phi \Rightarrow \Delta_{\texttt{bool}} \quad \vdash C_1 \sim C'_1 : \Phi \Rightarrow \Phi' \quad \vdash C_2 \sim C'_2 : \Phi \Rightarrow \Phi'}{\vdash (\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2) \sim (\texttt{if } B' \texttt{ then } C'_1 \texttt{ else } C'_2) : \Phi \Rightarrow \Phi'} \text{ [D-If]}$$

Figure 7: Core DDCC System

Basic equations Our first set of transformation rules express universally applicable structural equivalences for `while`-programs, without requiring any of the extra information gathered by the analysis.

Sequential unit laws:

$$\frac{\vdash C : \Phi \Rightarrow \Phi'}{\vdash (\texttt{skip};C) \sim C : \Phi \Rightarrow \Phi'}\ [\text{D-SU1}]$$

$$\frac{\vdash C : \Phi \Rightarrow \Phi'}{\vdash (C;\texttt{skip}) \sim C : \Phi \Rightarrow \Phi'}\ [\text{D-SU2}]$$

Associativity:

$$\frac{\vdash (C_1;C_2);C_3 : \Phi \Rightarrow \Phi'}{\vdash ((C_1;C_2);C_3) \sim (C_1;(C_2;C_3)) : \Phi \Rightarrow \Phi'}$$

In practice, one usually identifies programs up to associativity of sequential composition, rather than making explicit use of the rule above.

Commuting conversion for conditional:

$$\frac{\vdash \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \Phi \Rightarrow \Phi' \quad \vdash C_3 : \Phi' \Rightarrow \Phi''}{\begin{array}{ll}\vdash & (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2);C_3 \\ \sim & \texttt{if}\ B\ \texttt{then}\ (C_1;C_3)\ \texttt{else}\ (C_2;C_3) : \Phi \Rightarrow \Phi''\end{array}}\ [\text{D-CC}]$$

Loop unrolling:

$$\frac{\vdash \texttt{while}\ B\ \texttt{do}\ C : \Phi \Rightarrow \Phi' \qquad [\text{D-LU1}]}{\begin{array}{ll}\vdash & \texttt{while}\ B\ \texttt{do}\ C \\ \sim & \texttt{if}\ B\ \texttt{then}\ C;(\texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else skip} : \Phi \Rightarrow \Phi'\end{array}}$$

$$\frac{\vdash \texttt{while}\ B\ \texttt{do}\ C : \Phi \Rightarrow \Phi' \qquad [\text{D-LU2}]}{\begin{array}{ll}\vdash & \texttt{while}\ B\ \texttt{do}\ C \\ \sim & \texttt{while}\ B\ \texttt{do}\ (C;\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else skip}) : \Phi \Rightarrow \Phi'\end{array}}$$

Self-assignment elimination:

$$\vdash X\texttt{:=}X \sim \texttt{skip} : \Phi, X : \phi_{\texttt{int}} \Rightarrow \Phi, X : \phi_{\texttt{int}}\ [\text{D-SAs}]$$

In conjunction with the core rules, the rules above can be used to derive many of the basic equalities one might expect.[11] From a pragmatic point of view, however, they are somewhat unwieldy: even very simple proofs get quite large,

[11] Though the rules presented are in no sense complete. There are sound rules (arithmetic identities and equivalences for nested conditionals, for example) which are not consequences of the ones we have given.

with many applications of the symmetry and transitivity rules and many repeated sub-derivations. Reformulating the rules as logically equivalent versions which can be applied in more general contexts helps immensely. For example, a better formulation of one of the `skip` rules is the following:

$$\frac{\vdash C \sim \texttt{skip} : \Phi \Rightarrow \Phi' \quad \vdash C' \sim C'' : \Phi' \Rightarrow \Phi''}{\vdash (C\,;C') \sim C'' : \Phi \Rightarrow \Phi''}\ \text{[D-SU1']}$$

Presenting rules in this style is essentially trying to produce a system with a kind of cut-elimination property, but we leave serious consideration of proof-theoretic matters to future work.

Optimizing Transformations In this section we consider some more interesting rules, in which equations are predicated on information in the type system.

Dead assignment elimination:

$$\vdash (X\,\texttt{:=}E) \sim \texttt{skip} : \Phi, X : \phi_{\texttt{int}} \Rightarrow \Phi, X : \mathbb{T}_{\texttt{int}}\ \text{[D-DAs]}$$

Intuitively, the dead assignment rule says that an assignment to a variable is equivalent to `skip` *if* we are in a context in which the value of that variable does not matter.

Equivalent branches for conditional:

$$\frac{\vdash C_1 \sim C_2 : \Phi \Rightarrow \Phi'}{\vdash \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \sim C_1 : \Phi \Rightarrow \Phi'}\ \text{[D-BrE]}$$

An alternative form of this rule, which is a bit prettier, is

$$\frac{\vdash C_1 \sim C : \Phi \Rightarrow \Phi' \quad \vdash C_2 \sim C : \Phi \Rightarrow \Phi'}{\vdash \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \sim C : \Phi \Rightarrow \Phi'}\ \text{[D-BrE']}$$

Constant folding:

$$\frac{\vdash F_\tau : \Phi \Rightarrow \{c\}_\tau}{\vdash F_\tau \sim \texttt{c} : \Phi \Rightarrow \{c\}_\tau}\ \text{[D-CF]}$$

Known branch:

$$\frac{\vdash B : \Phi \Rightarrow \{\mathit{true}\} \quad \vdash C_1 \sim C' : \Phi \Rightarrow \Phi'}{\vdash (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2) \sim C' : \Phi \Rightarrow \Phi'}\ \text{[D-KBT]}$$

$$\frac{\vdash B : \Phi \Rightarrow \{\mathit{false}\} \quad \vdash C_2 \sim C' : \Phi \Rightarrow \Phi'}{\vdash (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2) \sim C' : \Phi \Rightarrow \Phi'}\ \text{[D-KBF]}$$

Dead while:

$$\frac{\vdash B : \Phi \Rightarrow \{false\}}{\vdash (\texttt{while } B \texttt{ do } C) \sim \texttt{skip} : \Phi \Rightarrow \Phi}\ [\text{D-DWh}]$$

This is actually derivable using loop unrolling [D-LU1] and known branch [D-KBF].

Divergence:

$$\frac{\vdash B : \Phi \Rightarrow \{true\} \quad \vdash C : \Phi \Rightarrow \Phi}{\vdash (\texttt{while } B \texttt{ do } C) : \Phi \Rightarrow \Phi'}\ [\text{D-Div}]$$

The type Φ' in the conclusion of the rule above is arbitrary because the loop will diverge when executed in any state in the domain of Φ.

The following is an easy induction, relying on Lemmas 4 and 5:

Theorem 3. *Assuming the abstract operations satisfy the correctness condition given in Definition 1, the core DDCC rules of Figure 7 and the additional rules of Section 3.2.3 are all sound:*

$$\begin{array}{rcl} \vdash F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau & \Longrightarrow & \models F_\tau \sim F'_\tau : \Phi \Rightarrow \phi_\tau \\ \vdash C \sim C' : \Phi \Rightarrow \Phi' & \Longrightarrow & \models C \sim C' : \Phi \Rightarrow \Phi' \end{array}$$

3.2.4 Example Transformations

These rules are sufficient to capture some non-trivial transformations, including constant propagation, dead-code elimination and program slicing [52]. Some example derivations are shown in Figure 8. We leave it as an exercise to prove larger examples, such as the slicing transformation:

```
I := 1;                  I := 1;
S := 0;
P := 1;                  P := 1
while I<N do (     ==>   while I<N do (
 S := S+I;
 P := P*I;                P := P*I;
 I := I+1;)               I := I+1;)
```

at type $N : \Delta_{\texttt{int}} \Rightarrow P : \Delta_{\texttt{int}}$. Here we expressed the fact that we were only interested in the final value of `P` simply by transforming it at a result type which only constrains the value of that variable to be preserved – all the others (in particular `S`) are typed at $\mathbb{T}_{\texttt{int}}$ and so are allowed to take any value.

Proof. Let $\Phi_0 = I : \mathbb{T}_{\texttt{int}}, S : \mathbb{T}_{\texttt{int}}, P : \mathbb{T}_{\texttt{int}}, N : \Delta_{\texttt{int}}$ and $\Phi_1 = I : \Delta_{\texttt{int}}, S : \mathbb{T}_{\texttt{int}}, P : \mathbb{T}_{\texttt{int}}, N : \Delta_{\texttt{int}}$. Then

$$\frac{\dfrac{\vdash \texttt{1} \sim \texttt{1} : \Phi_0 \Rightarrow \{1\} \quad \{1\} \leq \Delta_{\texttt{int}}}{\vdash \texttt{1} \sim \texttt{1} : \Phi_0 \Rightarrow \Delta_{\texttt{int}}}\ [\text{D-ESub}]}{\vdash \texttt{I :=1} \sim \texttt{I :=1} : \Phi_0 \Rightarrow \Phi_1}\ [\text{D-Ass}]$$

Constants, known branches and dead code:

$\mathcal{D}_1$:

$$\dfrac{\dfrac{\dfrac{}{\vdash X:\ \Phi, X:\{3\} \Rightarrow \{3\}}\,[\text{D-V}] \qquad \dfrac{}{\vdash \mathtt{3}:\ \Phi, X:\{3\}\Rightarrow\{3\}}\,[\text{D-N}]}{\vdash X = \mathtt{3}:\ \Phi, X:\{3\}\Rightarrow\{true\}}\,[\text{D-=}] \qquad \dfrac{\dfrac{}{\vdash \mathtt{7}:\ \Phi,X:\{3\}\Rightarrow\{7\}}\,[\text{D-N}]}{\vdash X\mathtt{:=7}:\ \Phi,X:\{3\}\Rightarrow\Phi,X:\{7\}}\,[\text{D-Ass}]}{\vdash (\mathtt{if}\ X=\mathtt{3}\ \mathtt{then}\ X\mathtt{:=7}\ \mathtt{else}\ \mathtt{skip}) \sim (X\mathtt{:=7}) : \Phi, X:\{3\}\Rightarrow \Phi,X:\{7\}}\,[\text{D-KBT}]$$

$\mathcal{D}_2$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash X:\Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \{7\}}\,[\text{D-V}] \qquad \dfrac{}{\vdash \mathtt{1}:\Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \{1\}}\,[\text{D-N}]}{\vdash X+\mathtt{1}:\Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \{8\}}\,[\text{D-+}]}{\vdash X+\mathtt{1}\sim\mathtt{8}:\Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \{8\}}\,[\text{D-CF}]}{\vdash \begin{array}{l} Z\mathtt{:=}X+\mathtt{1} \\ \sim Z\mathtt{:=8}\end{array} : \Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\{7\}, Z:\{8\}}\,[\text{D-Ass}] \qquad \Phi, X:\{7\}, Z:\{8\}\ \leq\ \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}{\vdash \begin{array}{l} Z\mathtt{:=}X+\mathtt{1} \\ \sim Z\mathtt{:=8}\end{array} : \Phi, X:\{7\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}\,[\text{D-CSub}]$$

$\mathcal{D}_3$:

$$\dfrac{\dfrac{}{\vdash (X\mathtt{:=7})\sim\mathtt{skip}:\Phi, X:\{3\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\mathbb{T}_{\mathtt{int}}}\,[\text{D-DAss}] \qquad \dfrac{\vdash (\mathtt{8}):\Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \{8\}}{\vdash Z\mathtt{:=8}:\Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}}{\vdash (X\mathtt{:=7;}Z\mathtt{:=8})\sim Z\mathtt{:=8}:\Phi, X:\{3\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}\,[\text{D-SU1'}]$$

$$\dfrac{\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash \begin{array}{l} (\mathtt{if}\ X=\mathtt{3}\ \mathtt{then}\ X\mathtt{:=7}\ \mathtt{else}\ \mathtt{skip;}Z\mathtt{:=}X+\mathtt{1}) \\ \sim (X\mathtt{:=7;}Z\mathtt{:=8})\end{array} : \Phi, X:\{3\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}\,[\text{D-Seq}] \qquad \mathcal{D}_3}{\vdash \begin{array}{l} (\mathtt{if}\ X=\mathtt{3}\ \mathtt{then}\ X\mathtt{:=7}\ \mathtt{else}\ \mathtt{skip;}Z\mathtt{:=}X+\mathtt{1}) \\ \sim (Z\mathtt{:=8})\end{array} : \Phi, X:\{3\}, Z:\mathbb{T}_{\mathtt{int}} \Rightarrow \Phi, X:\mathbb{T}_{\mathtt{int}}, Z:\{8\}}\,[\text{D-CTr}]$$

Figure 8: Examples of DDCC Transformations

Then as
$$\vdash \texttt{S := 0} \sim \texttt{skip} : \Phi_1 \Rightarrow \Phi_1 \text{ [D-DAs]}$$
We can deduce
$$\vdash (\texttt{I := 1;S := 0}) \sim (\texttt{I := 1}) : \Phi_0 \Rightarrow \Phi_1$$
by [D-SU2']. Now let $\Phi_2 = I : \Delta_{\texttt{int}}, S : \mathbb{T}_{\texttt{int}}, P : \Delta_{\texttt{int}}, N : \Delta_{\texttt{int}}$ and similar reasoning yields
$$\vdash (\texttt{I := 1;S := 0;P:=1}) \sim (\texttt{I := 1;P:=1}) : \Phi_0 \Rightarrow \Phi_2$$
Next we show
$$\vdash \texttt{I} < \texttt{N} \sim \texttt{I} < \texttt{N} : \Phi_2 \Rightarrow \Delta_{\texttt{bool}}$$
and
$$\vdash (\texttt{S} := \texttt{S} + \texttt{1}; \texttt{P} := \texttt{P} * \texttt{I}; \texttt{I} := \texttt{I} + \texttt{1}) \sim (\texttt{P} := \texttt{P} * \texttt{I}; \texttt{I} := \texttt{I} + \texttt{1}) : \Phi_2 \Rightarrow \Phi_2$$
so that, by [D-Whl]:
$$\vdash \begin{array}{l} (\texttt{while I} < \texttt{N do } (\texttt{S} := \texttt{S} + \texttt{1}; \texttt{P} := \texttt{P} * \texttt{I}; \texttt{I} := \texttt{I} + \texttt{1})) \sim \\ (\texttt{while I} < \texttt{N do } (\texttt{P} := \texttt{P} * \texttt{I}; \texttt{I} := \texttt{I} + \texttt{1})) \end{array} : \Phi_2 \Rightarrow \Phi_2$$
Then as $N : \Delta_{\texttt{int}} \leq \Phi_0$ and $\Phi_2 \leq P : \Delta_{\texttt{int}}$ we can plug the bits together with [D-Seq] and [D-CSub] and we're done. $\square$

3.2.5 Secure Information Flow

It is worth observing that the $\mathbb{T}$,Δ fragment of our calculus can be seen as a non-interference type system. Figure 9 presents a version of a type system for secure information flow due to Smith and Volpano [45]. In this system, a *security level*, σ, is either low (L) or high (H). A context γ is then a finite map from variables to security levels:
$$\gamma \quad := \quad - \mid \gamma, X : \sigma_{\texttt{int}}$$
Given such a context, the type system assigns a security level ($\sigma_{\texttt{int}}$ or $\sigma_{\texttt{bool}}$) to each expression and ($\sigma_{\texttt{com}}$) to each command. The property which the type system ensures is that any typeable command does not allow information to flow (either directly, via assignment, or indirectly, via control flow) from high security variables to low security ones. We define a translation $(\cdot)^*$ from the Smith/Volpano system into DDCC as follows:

Expression types: $L^*_\tau = \Delta_\tau$ and $H^*_\tau = \mathbb{T}_\tau$.

Contexts: $-^* = -$ and $(\gamma, X : \sigma_{\texttt{int}})^* = \gamma^*, X : \sigma^*_{\texttt{int}}$.

Judgements:
$$\begin{array}{rcl} (\gamma \vdash F : \sigma_\tau)^* & = & \vdash F \sim F : \gamma^* \Rightarrow \sigma^*_\tau \\ (\gamma \vdash C : L_{\texttt{com}})^* & = & \vdash C \sim C : \gamma^* \Rightarrow \gamma^* \\ (\gamma \vdash C : H_{\texttt{com}})^* & = & \vdash C \sim \texttt{skip} : \gamma^* \Rightarrow \gamma^* \end{array}$$

$$\gamma, X : \sigma_{\mathtt{int}} \vdash X : \sigma_{\mathtt{int}} \qquad \gamma \vdash \mathtt{n} : \sigma_{\mathtt{int}} \qquad \gamma \vdash \mathtt{b} : \sigma_{\mathtt{bool}}$$

$$\frac{\gamma \vdash E : \sigma_{\mathtt{int}} \quad \gamma \vdash E' : \sigma_{\mathtt{int}}}{\gamma \vdash E \ \mathtt{iop} E' \ : \sigma_{\mathtt{int}}} + \text{similar for } \textit{bop} \text{ and } \textit{lop}$$

$$\frac{\gamma, X : \sigma_{\mathtt{int}} \vdash E : \sigma_{\mathtt{int}}}{\gamma, X : \sigma_{\mathtt{int}} \vdash X \mathtt{:=} E : \sigma_{\mathtt{com}}}$$

$$\frac{\gamma \vdash C : \sigma_{\mathtt{com}} \quad \gamma \vdash C' : \sigma_{\mathtt{com}}}{\gamma \vdash C \mathtt{;} C' : \sigma_{\mathtt{com}}} \qquad \frac{\gamma \vdash B : \sigma_{\mathtt{bool}} \quad \gamma \vdash C : \sigma_{\mathtt{com}} \quad \gamma \vdash C' : \sigma_{\mathtt{com}}}{\gamma \vdash \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C' : \sigma_{\mathtt{com}}}$$

$$\frac{\gamma \vdash B : L_{\mathtt{bool}} \quad \gamma \vdash C : L_{\mathtt{com}}}{\gamma \vdash \mathtt{while}\ B\ \mathtt{do}\ C : L_{\mathtt{com}}} \qquad \frac{\gamma \vdash F : L_\tau}{\gamma \vdash F : H_\tau} \qquad \frac{\gamma \vdash C : H_{\mathtt{com}}}{\gamma \vdash C : L_{\mathtt{com}}}$$

Figure 9: Smith/Volpano Type System

Theorem 4. *For any judgement J derivable in the Smith/Volpano system, J^* is derivable in DDCC*

Definition 2. *In the context of a security type assignment γ, a command C satisfies* strong sequential noninterference *if* $\models C \sim C : \gamma^* \Rightarrow \gamma^*$.

This version of non-interference is the semantic security property intended by Smith and Volpano, though the actual property established by the soundness proof in [45] is more syntactic and intensional, as it is defined in terms of their particular typing rules. Our notion of intereference is *strong* because it is termination-sensitive: varying the high-security inputs affects neither the low-security outputs *nor* the termination behaviour. In the absence of any termination analysis, this is enforced by the rather brutal approach of making all high-security commands total. The weaker notion of non-interference that is achieved by the earlier system of Volpano, Smith and Irvine [47] does not seem to translate directly into DDCC.

Even without constant tracking, DDCC is marginally more powerful than the Smith/Volpano system. For example, if `H` is a high-security variable, and `L` is low-security then the following are easily shown to satisfy non-interference in DDCC, but would be rejected by the Smith/Volpano system:

1. `if H > 3 then H := L ; L := 1 else L := 1`
2. `L := H ; L := 3`

3.3 Relational Hoare Logic

There are many common optimizing transformations which are not captured by DDCC. In particular:

- It does not capture any transformations that take advantage of the fact that one knows statically which way a boolean test must have evaluated if one is within a particular branch of a conditional, or either in the body of or have just left a `while`-loop. For example, the judgement

$$\begin{gathered}\vdash (\texttt{if } X = \texttt{3} \texttt{ then } Y\texttt{:=}X \texttt{ else } Y\texttt{:=3}) \\ \sim (Y\texttt{:=3}) : X : \Delta \Rightarrow Y : \{3\}\end{gathered}$$

  is semantically valid but not derivable.

- It cannot express the preservation of the values of expressions, except where they are statically known to be a particular constant. These means even trivial code-motion transformations cannot be derived.

Rather than making piecemeal additions to the system to address such particular weaknesses, we instead now jump straight to the presentation of a rather more general system, which we call Relational Hoare Logic (RHL), into which many of these extensions or alternative type systems can be embedded.

Unlike DDCC, RHL does not look like a conventional type-based analysis system – it has a rather general syntax for relations and is parameterized on some system for deciding the entailment relation between them. The intention is that more specific analyses and transformations can be formulated as subsystems of RHL by restricting the syntax of assertions and providing particular approximations to the entailment relation. Another way in which RHL goes beyond DDCC is that it is not restricted to partial equivalence relations, which deserves some comment.

PERs are certainly privileged: they are the basis of equational reasoning, and we will nearly always be trying to prove that one program phrase is related to another by a PER so that we can perform a rewrite in some context. However, in order to establish that two phrases are related by a PER, we often have to do some local reasoning using more general relations. This is familiar in the semantics of polymorphic type theories: types are interpreted by PERs, and polymorphism by quantification over PERs, but parametricity theorems and equivalence results for implementations of abstract dataypes arise from substituting more general relations. To give some intuition for why this might be so, consider proving the equivalence

```
X := -Y;            X := Y;
Z := Z-X;     ==>   Z := Z+X;
X := -X;
```

at, say, $Y : \Delta_{\texttt{int}}, Z : \Delta_{\texttt{int}} \Rightarrow X : \Delta_{\texttt{int}}, Y : \Delta_{\texttt{int}}, Z : \Delta_{\texttt{int}}$. If we try to to establish that the two commands are related by this PER by relating their intermediate states (though this is not the only approach one could take), we will need to use the relation that the value of `X` in one state is the negation of that in the other, which is not a PER.

RHL is an extremely simple variation on traditional Floyd-Hoare logic [22]. Instead of assertions which denote predicates on states and judgements which say that terminating execution of a command in a state satisfying a precondition will yield a state satisfying a postcondition, we directly axiomatise when a *pair* of commands map a given pre-*relation* into a given post-*relation*. Binary relations on states are simply specified by boolean expressions of the language over variables tagged with an indication of which of the two states they refer to. At first sight, this may seem frighteningly simple-minded, but it actually works rather nicely. In this presentation we do not consider quantification over metavariables (also known as *auxiliary*, or *ghost*, variables); their addition does add power to the logic, but simple global analyses seem to be expressible without them.

3.4 RHL Syntax and Semantics

3.4.1 Syntax

We define generalized expressions and relational assertions as follows:

$$\begin{array}{rcl} \texttt{gexp} \ni GE & := & \texttt{n} \mid X\langle 1\rangle \mid X\langle 2\rangle \mid GE \ \texttt{iop}\ GE \\ \texttt{relexp} \ni \Phi & := & \texttt{b} \mid GE\ \texttt{bop}\ GE \mid \texttt{not}\Phi \mid \Phi\ \texttt{lop}\ \Phi \end{array}$$

We overload the notation $(\cdot)\langle 1\rangle$ and $(\cdot)\langle 2\rangle$ to stand for homomorphic embeddings `int exp` $\rightarrow$ `gexp` and `bool exp` $\rightarrow$ `relexp` in the obvious way. The basic judgement form is $\vdash C \sim C' : \Phi \Rightarrow \Phi'$ (though the use of $\sim$ for arbitrary relations is arguably bad).

Semantics The semantics of generalized expressions as integer-valued functions of two states, and of relational assertions as relations on states are just as one would expect:

$$\begin{array}{rcl} [\![GE]\!] & \in & \mathbb{S}\times\mathbb{S}\rightarrow\mathbb{Z} \\ [\![\texttt{n}]\!](S_1,S_2) & = & n \\ [\![X\langle 1\rangle]\!](S_1,S_2) & = & S_1(X) \\ [\![X\langle 2\rangle]\!](S_1,S_2) & = & S_2(X) \\ [\![E\ \texttt{iop}\ F]\!](S_1,S_2) & = & ([\![E]\!](S_1,S_2))\ \ iop\ \ ([\![F]\!](S_1,S_2)) \end{array}$$

$$\begin{array}{rcl} [\![\Phi]\!] & \subseteq & \mathbb{S}\times\mathbb{S} \\ & = & \{(S,S') \mid \chi_\Phi(S,S') = true\} \\ \chi_{\texttt{true}}(S'S') & = & true \\ \chi_{\texttt{false}}(S,S') & = & false \\ \chi_{E\ \texttt{bop}\ F}(S,S') & = & [\![E]\!](S,S')\ \ bop\ \ [\![F]\!](S,S') \\ \chi_{\Phi\ \texttt{lop}\ \Phi'}(S,S') & = & \chi_\Phi(S,S')\ \ lop\ \ \chi_{\Phi'}(S,S') \\ \chi_{\texttt{not}\Phi}(S,S') & = & \neg(\chi_\Phi(S,S')) \end{array}$$

$$\vdash \texttt{skip} \sim \texttt{skip} : \Phi \Rightarrow \Phi \text{ [R-Skip]}$$

$$\frac{\vdash C \sim C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi' \qquad \vdash D \sim D' : \Phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle) \Rightarrow \Phi'}{\vdash \texttt{if } B \texttt{ then } C \texttt{ else } D \sim \texttt{if } B' \texttt{ then } C' \texttt{ else } D' : \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi'} \text{ [R-If]}$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi' \qquad \vdash D \sim D' : \Phi' \Rightarrow \Phi''}{\vdash C \texttt{ ; } D \sim C' \texttt{ ; } D' : \Phi \Rightarrow \Phi''} \text{ [R-Seq]}$$

$$\vdash X \texttt{ := } E \sim Y \texttt{ := } E' : \Phi[E\langle 1\rangle / X\langle 1\rangle, E'\langle 2\rangle / Y\langle 2\rangle] \Rightarrow \Phi \text{ [R-Ass]}$$

$$\frac{\vdash C \sim C' : \Phi \wedge (B\langle 1\rangle \wedge B'\langle 2\rangle) \Rightarrow \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle)}{\vdash \texttt{while } B \texttt{ do } C \sim \texttt{while } B' \texttt{ do } C' : \Phi \wedge (B\langle 1\rangle = B'\langle 2\rangle) \Rightarrow \Phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle)} \text{ [R-Whl]}$$

$$\frac{\vdash C \sim C' : \Phi_1 \Rightarrow \Phi_2 \qquad \models \Phi_1' \leq \Phi_1 \qquad \models \Phi_2 \leq \Phi_2'}{\vdash C \sim C' : \Phi_1' \Rightarrow \Phi_2'} \text{ [R-Sub]}$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi' \qquad \models PER(\Phi \Rightarrow \Phi')}{\vdash C' \sim C : \Phi \Rightarrow \Phi'} \text{ [R-Sym]}$$

$$\frac{\vdash C \sim C' : \Phi \Rightarrow \Phi' \qquad \vdash C' \sim C'' : \Phi \Rightarrow \Phi' \qquad \models PER(\Phi \Rightarrow \Phi')}{\vdash C \sim C'' : \Phi \Rightarrow \Phi'} \text{ [R-Tr]}$$

Figure 10: Core Relational Hoare Logic

The intended meaning of judgements is given by

$$\begin{array}{rl} & \models C \sim C' : \Phi \Rightarrow \Phi' \\ \equiv & \forall (S_1, S_2) \in [\![\Phi]\!].\ ([\![C]\!](S_1), [\![C']\!](S_2)) \in [\![\Phi']\!]_\perp \end{array}$$

We will also need some auxiliary semantic judgements, whose meanings are as follows:

$$\begin{array}{rcl} \models \Phi \leq \Phi' & \equiv & [\![\Phi]\!] \subseteq [\![\Phi']\!] \\ \models PER(\Phi) & \equiv & ([\![\Phi]\!] \circ [\![\Phi]\!] \subseteq [\![\Phi]\!]) \text{ and } ([\![\Phi]\!]^{-1} \subseteq [\![\Phi]\!]) \end{array}$$

Inference Rules The core rules for RHL are shown in Figure 10. Observe that, as was the case in DDCC, the basic rules ensure that the same conditional branches are taken and that loops are executed the same number of times on the two sides. Note also that one could add distinct semantic judgements for symmetry and transitivity, rather than requiring both. The assignment rule is surprisingly liberal, but there is no reason to require the assigned variables to be the same in both commands.

3.4.2 Equations

As with DDCC, we will specify optimizing transformations by adding extra (sound) rules to the core. But even before we do that, RHL can justify some useful transformations. Here's an example of removing a redundant evaluation:

1. $\vdash \begin{array}{l} Z\texttt{:=}Y\texttt{+1} \\ \sim Z\texttt{:=}X \end{array} : \begin{array}{l} X\langle 1\rangle = X\langle 2\rangle \wedge \\ Y\langle 1\rangle + 1 = X\langle 2\rangle \end{array} \Rightarrow \begin{array}{l} X\langle 1\rangle = X\langle 2\rangle \wedge \\ Z\langle 1\rangle = Z\langle 2\rangle \end{array}$ by [R-Ass]

2. $\vdash \begin{array}{l} X\texttt{:=}Y\texttt{+1} \\ \sim X\texttt{:=}Y\texttt{+1} \end{array} : \begin{array}{l} Y\langle 1\rangle + 1 = Y\langle 2\rangle + 1 \wedge \\ Y\langle 1\rangle + 1 = Y\langle 2\rangle + 1 \end{array} \Rightarrow \begin{array}{l} X\langle 1\rangle = X\langle 2\rangle \wedge \\ Y\langle 1\rangle + 1 = X\langle 2\rangle \end{array}$ by [R-Ass]

3. $\models (Y\langle 1\rangle = Y\langle 2\rangle) \leq \begin{array}{l} Y\langle 1\rangle + 1 = Y\langle 2\rangle + 1 \wedge \\ Y\langle 1\rangle + 1 = Y\langle 2\rangle + 1 \end{array}$ by logic

4. $\vdash \begin{array}{l} X\texttt{:=}Y\texttt{+1} \\ \sim X\texttt{:=}Y\texttt{+1} \end{array} : Y\langle 1\rangle = Y\langle 2\rangle \Rightarrow \begin{array}{l} X\langle 1\rangle = X\langle 2\rangle \wedge \\ Y\langle 1\rangle + 1 = X\langle 2\rangle \end{array}$ by [R-Sub] applied to 2. and 3.

5. $\vdash \begin{array}{l} X\texttt{:=}Y\texttt{+1;}Z\texttt{:=}Y\texttt{+1} \\ \sim X\texttt{:=}Y\texttt{+1;}Z\texttt{:=}X \end{array} : Y\langle 1\rangle = Y\langle 2\rangle \Rightarrow \begin{array}{l} X\langle 1\rangle = X\langle 2\rangle \wedge \\ Z\langle 1\rangle = Z\langle 2\rangle \end{array}$ by [R-Seq] applied to 4. and 1.

Basic Equations The basic equations we presented in the context of DDCC are still valid for RHL, with the exception of self-assignment elimination, though the contextual versions are now more powerful than the simple ones, so we take [R-SU1'L] and [R-SU2'L] (and their symmetric versions) as basic.

I believe an RHL version of [D-CC] is probably admissible given the other RHL rules presented here, but admissibility is not preserved by adding further rules, so it seems easiest to add something explicit, at least to make sure that the DDCC embedding theorem is valid. There are a number of choices for natural RHL rules which imply [D-CC]. The following is one:

$$\frac{\vdash \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \sim C : \Phi \Rightarrow \Phi'}{\vdash C_1 \sim C : \Phi \wedge B\langle 1\rangle \Rightarrow \Phi'} \text{[R-CBInvTL]}$$

$$\frac{\vdash \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \sim C : \Phi \Rightarrow \Phi'}{\vdash C_2 \sim C : \Phi \wedge \texttt{not} B\langle 1\rangle \Rightarrow \Phi'} \text{[R-CBInvFL]}$$

Plus the obvious ones the other way around. These are the inverted versions of [R-CB] (see later).

We also add the loop unrolling rules [R-LU1L] and [R-LU2L] and their right-handed versions, mainly to ensure that the embedding of DDCC in RHL works. A more powerful inductive rules could be used here instead. It should also be remarked that the messy business of having left and right variants of rules can also be avoided by use of a Φ^{-1} rule, which swaps 1 and 2. (There's also a case to be made for adding syntactic relational composition.)

Optimizing Transformations

Falsity:

$$\vdash C \sim C' : \texttt{false} \Rightarrow \Phi \text{ [R-F]}$$

Dead assignment:

$$\vdash X\texttt{:=}E \sim \texttt{skip} : \Phi[E\langle 1\rangle/X\langle 1\rangle] \Rightarrow \Phi \text{ [R-DAssL]}$$

$$\vdash \texttt{skip} \sim X\texttt{:=}E : \Phi[E\langle 2\rangle/X\langle 2\rangle] \Rightarrow \Phi \text{ [R-DAssR]}$$

These rules subsume our previous dead-assignment and self-assignment rules. With the basic rules for skip, they subsume the [R-Ass] rule too.

Common branch:

$$\frac{\vdash C \sim D : \Phi \wedge B\langle 1\rangle \Rightarrow \Phi' \quad \vdash C' \sim D : \Phi \wedge \texttt{not} B\langle 1\rangle \Rightarrow \Phi'}{\vdash \texttt{if } B \texttt{ then } C \texttt{ else } C' \sim D : \Phi \Rightarrow \Phi'} \text{[R-CBL]}$$

Plus a version with the conditional on the right. These subsume our earlier equivalent branch rule, and (via the falsity equation) the known-branch rules and the [R-If] rule.

Dead while:

$$\vdash \texttt{while } B \texttt{ do } C \sim \texttt{skip} : \Phi \wedge \texttt{not} B\langle 1\rangle \Rightarrow \Phi \wedge \texttt{not} B\langle 1\rangle \text{ [R-DWhl]}$$

Plus the variant with `skip` on the left.

Soundness follows by induction:

Theorem 5. *For all C,C',Φ,Φ', if $\vdash C \sim C' : \Phi \Rightarrow \Phi'$ is derivable using the rules in Figure 10 and Section 3.4.2 then $\models C \sim C' : \Phi \Rightarrow \Phi'$.*

3.4.3 Examples

With these rules, one can prove the correctness of many traditional compiler optimizations, including various forms of code motion and predicated transformation. Producing proofs in RHL is fairly straightforward, so we just give a couple of small examples of the sort of thing one can prove.

Invariant hoisting:

```
while I<N do            X := Y+1;
  X := Y+1;     ==>     while I<N do
  I := I+X;               I := I+X;
```

at type $\Phi \Rightarrow \Phi$ where Φ is $I\langle 1\rangle = I\langle 2\rangle \wedge N\langle 1\rangle = N\langle 2\rangle \wedge Y\langle 1\rangle = Y\langle 2\rangle$. Note that the lifting is only valid because we do not care about the final value of X. The proof makes two uses of the dead-assignment rule, which is a common pattern for performing code-motion in RHL: one effectively adds `skip`s to both sides to make them the same 'shape', shows the equivalence using the congruence rules and then removes the `skip`s.

Proof. Let Φ be as defined above, $B = (I < N)$ and $\Phi' = \Phi \wedge (X\langle 2\rangle = Y\langle 2\rangle + 1) \wedge (B\langle 1\rangle = B\langle 2\rangle)$. Now by [R-Ass]

$$\vdash \texttt{I := I+X} \sim \texttt{I := I+X} : \Phi'[(I\langle 1\rangle + X\langle 1\rangle)/I\langle 1\rangle, (I\langle 2\rangle + X\langle 2\rangle)/X\langle 2\rangle] \Rightarrow \Phi'$$

and by [R-DAs]

$$\vdash \begin{array}{l}\texttt{X := Y+1}\\ \sim \texttt{skip}\end{array} : \begin{array}{l}\Phi'[(I\langle 1\rangle + X\langle 1\rangle)/I\langle 1\rangle, (I\langle 2\rangle + X\langle 2\rangle)/X\langle 2\rangle][(Y\langle 1\rangle + 1)/X\langle 1\rangle]\\ \Rightarrow \Phi'[(I\langle 1\rangle + X\langle 1\rangle)/I\langle 1\rangle, (I\langle 2\rangle + X\langle 2\rangle)/X\langle 2\rangle]\end{array}$$

so by [R-SU1'L]

$$\begin{array}{l}\vdash (\texttt{X := Y+1; I := I+X}) \sim (\texttt{I := I+X})\\ \quad : \Phi'[(I\langle 1\rangle + X\langle 1\rangle)/I\langle 1\rangle, (I\langle 2\rangle + X\langle 2\rangle)/X\langle 2\rangle][(Y\langle 1\rangle + 1)/X\langle 1\rangle] \Rightarrow \Phi'\end{array}$$

Expanding the substitution on the left gives

$$\begin{array}{rcl}\Phi'' & = & (I\langle 1\rangle + Y\langle 1\rangle + 1) = (I\langle 2\rangle + X\langle 2\rangle)\\ & \wedge & N\langle 1\rangle = N\langle 2\rangle\\ & \wedge & Y\langle 1\rangle = Y\langle 2\rangle\\ & \wedge & X\langle 2\rangle = Y\langle 2\rangle + 1\\ & \wedge & ((I\langle 1\rangle + (Y\langle 1\rangle + 1)) < N\langle 1\rangle) = (I\langle 2\rangle + X\langle 2\rangle < N\langle 2\rangle)\end{array}$$

Logic and arithmetic then give $\models (\Phi \wedge (X\langle 2\rangle = Y\langle 2\rangle + 1) \wedge B\langle 1\rangle \wedge B\langle 2\rangle) \leq \Phi''$, so by [R-Sub] and [R-Whl]

$$\begin{array}{l}\vdash \texttt{while I<N do (X :=Y+1;I:=I+X)} \sim \texttt{while I<N do I := I+X}\\ \quad : \Phi' \Rightarrow \Phi \wedge (X\langle 2\rangle = Y\langle 2\rangle + 1) \wedge \texttt{not}(B\langle 1\rangle \vee B\langle 2\rangle)\end{array}$$

Now by [R-DAs]

$$\texttt{skip} \sim \texttt{X := Y+1} : \Phi'[(Y\langle 2\rangle + 1)/X\langle 2\rangle] \Rightarrow \Phi'$$

and $\models \Phi \leq \Phi'[(Y\langle 2\rangle + 1)/X\langle 2\rangle]$. So by [R-Sub] and [R-SU1']

$$\vdash \texttt{while I<N do (X :=Y+1;I:=I+X)} \sim \texttt{X := Y+1; while I<N do I := I+X}$$
$$: \Phi \Rightarrow \Phi \wedge (X\langle 2\rangle = Y\langle 2\rangle + 1) \wedge \texttt{not}(B\langle 1\rangle \vee B\langle 2\rangle)$$

and clearly $\models (\Phi \wedge (X\langle 2\rangle = Y\langle 2\rangle + 1) \wedge \texttt{not}(B\langle 1\rangle \vee B\langle 2\rangle)) \leq \Phi$ so we're done by [R-Sub]. □

Sophisticated dead-code:

```
if X>3 then Y := X else Y := 7    ==> skip
```

at type $(X\langle 1\rangle = X\langle 2\rangle \wedge Y\langle 1\rangle > 2 \wedge Y\langle 2\rangle > 2) \Rightarrow (Y\langle 1\rangle > 2 \wedge Y\langle 2\rangle > 2)$. I.e. if all that matters about the value of Y in the rest of the derivation is that it is greater than 2, then the conditional has no effect.

Proof. Write Φ for $(X\langle 1\rangle = X\langle 2\rangle \wedge Y\langle 1\rangle > 2 \wedge Y\langle 2\rangle > 2)$ and Φ' for $(Y\langle 1\rangle > 2 \wedge Y\langle 2\rangle > 2)$. By [R-DAssL]

$$\vdash Y\texttt{:=}X \sim \texttt{skip} : \Phi'[X\langle 1\rangle/Y\langle 1\rangle] \Rightarrow \Phi'$$

and

$$\vdash Y\texttt{:=7} \sim \texttt{skip} : \Phi'[7/Y\langle 1\rangle] \Rightarrow \Phi'$$

It is then trivial to check

$$\models \Phi \wedge (X\langle 1\rangle > 3) \leq \Phi'[X\langle 1\rangle/Y\langle 1\rangle] \quad \text{and}$$
$$\models \Phi \wedge \texttt{not}(X\langle 1\rangle > 3) \leq \Phi'[7/Y\langle 1\rangle]$$

so that by two applications of [R-Sub] and one of [R-CBL] we are done. □

The main weakness of RHL as presented here relates to its treatment of loops. Since we insist that transformed programs have the same termination behaviour as the original, but have no non-trivial termination analysis, this is hardly suprising. I believe it is possible to add sound rules which can justify some cases of loop distribution/fusion, but more ambitious loop optimizations seem to require either a language with restricted iteration constructs or a logic which can reason about termination.

3.4.4 Embedding Simpler Logics in RHL

RHL is powerful but hardly suitable for direct implementation in a compiler. However, it can provide a useful framework for developing sound type and transformation systems which are more specific. One would start by identifying a restricted sublanguage of relational assertions. For example, several useful analyses can be formulated using only partial equivalence relations generated from axioms such as:

$$\vdash PER(E\langle 1\rangle = E\langle 2\rangle) \qquad \vdash PER(B\langle 1\rangle = B\langle 2\rangle)$$

$$\vdash PER(B\langle 1\rangle \wedge B\langle 2\rangle)$$

plus rules stating that PERs are closed under conjunction, disjoint union and the arrow constructor. Our earlier DDCC system is of this form, with state relations being formed as conjunctions of primitive assertions of the forms $X\langle 1\rangle = X\langle 2\rangle$ and $X\langle 1\rangle = n \wedge X\langle 2\rangle = n$. The rules of DDCC can then be presented as derived rules in RHL.

For $F, F' \in \tau$ `exp` and DDCC expression type ϕ_τ, define the RHL relation $(F \sim F' : \phi_\tau)^*$ as follows:

$$\begin{array}{rcl} (F \sim F : \mathbb{F})^* & = & \texttt{false} \\ (F \sim F' : \{c\})^* & = & (F\langle 1\rangle = \texttt{c}) \wedge (F'\langle 2\rangle = \texttt{c}) \\ (F \sim F' : \Delta)^* & = & (F\langle 1\rangle = F'\langle 2\rangle) \\ (F \sim F' : \mathbb{T})^* & = & \texttt{true} \end{array}$$

Then for a DDCC state type Φ, define the RHL relation Φ^* by

$$\begin{array}{rcl} (-)^* & = & \texttt{true} \\ (\Phi, X : \phi_{\texttt{int}})^* & = & \Phi^* \wedge (X \sim X : \phi_{\texttt{int}})^* \end{array}$$

Theorem 6. *For all $F, F', \Phi, \Phi', C, C', \phi$:*

1. *If $\vdash F \sim F' : \Phi \Rightarrow \phi$ then $\models \Phi^* \leq (F \sim F' : \phi)^*$.*
2. *If $\vdash C \sim C' : \Phi \Rightarrow \Phi'$ in DDCC then $\vdash C \sim C' : \Phi^* \Rightarrow \Phi'^*$ in RHL.*

A natural question is whether the usual Hoare logic can be embedded in RHL. One's first thought might be that a partial correctness judgement $\vdash \{P\}C\{Q\}$ would be equivalent to the 'squared' RHL judgement

$$\vdash C \sim C : P\langle 1\rangle \wedge P\langle 2\rangle \Rightarrow Q\langle 1\rangle \wedge Q\langle 2\rangle$$

but this is not the case because C's termination behaviour might differ on two states satisfying P. Nor can one simply intersect the pre- and post-relations with the identity relation on states, since we do not have syntax for that 'global' identity relation. If we fix C, however, we can conjoin the pre- and post-relations with $X\langle 1\rangle = X\langle 2\rangle$ for every variable X occurring in C and thus effectively recover Hoare logic.[12] Going the other way, one can soundly extend RHL with the squared versions of valid *total* correctness judgements $\vdash [P]C[Q]$.

As a simple, concrete example of the embedding approach, Figure 11 presents (a very naive version of) a type system AERC for available expression analysis and removal of redundant evaluation. State types Θ are finite sets $\{X_i = E_i \mid 1 \leq i \leq n\}$ of equalities between variables and expressions (in which the same variable may occur multiple times on the left) and we write $\Theta \leq \Theta'$ for $\Theta \supseteq \Theta'$. The macros *kill* and *gen* are defined by

$$\begin{array}{lcl} \mathit{kill}(\Theta, X) & = & \{(X_i = E_i) \in \Theta \mid X_i \neq X \wedge X \notin E_i\} \\ \mathit{gen}(X, E) & = & \begin{cases} \{X = E\} & \text{if } X \notin E \\ \{\} & \text{otherwise} \end{cases} \end{array}$$

The translation of the AERC into RHL is indexed by a finite set V of variables. Define

$$\Theta^*_V = \bigwedge_{X \in V} (X\langle 1\rangle = X\langle 2\rangle) \wedge \bigwedge_{(X=E)\in\Theta} (X\langle 1\rangle = E\langle 1\rangle)$$

It is easy to see that for any Θ, $\models PER(\Theta^*_V)$ and that $\Theta \leq \Theta'$ implies $\Theta^*_V \leq \Theta'^*_V$. The following asserts the soundness of the translation, and hence of AERC:

Theorem 7. *For any expressions E,F and commands C,D all of whose variables occur in V,*

1. *If $\vdash E \sim F : \Theta \Rightarrow \tau$ then $\models \Theta^*_V \leq (E\langle i\rangle = F\langle j\rangle)$ for $i, j \in \{1, 2\}$.*
2. *If $\vdash C \sim D : \Theta \Rightarrow \Theta'$ in AERC then $\vdash C \sim D : \Theta^*_V \Rightarrow \Theta'^*_V$ in RHL.*

3.5 Related Work

We have already mentioned the work of Wand et al and of Amtoft on proving soundness of optimizing transormations for functional languages. Other examples include Damiani and Giannini on dead-variables [17, 18], Kobayashi on dead-variables [28] and Benton and Kennedy on effects [12]. Damiani and Giannini explicitly use PERs in giving the semantics of their analysis system but give a more algorithmic account its use in transformation. Benton and Kennedy present optimizing transformations as equations in context, but derive those (rather clumsily) from a predicate-based semantics for the analysis.

Recently Lacey et al. [30] described how some of the classical [27, 21, 4] transformations considered here (dead code elimination, constant folding and a simple code-motion transformation) can be formulated as conditional rewrite rules on control flow graphs. The rewrites are predicated on temporal logic formulae expressing (intensionally) the contexts in which the rewrites may be applied. The authors then use a small-step operational semantics to verify that under these conditions, their transformations preserve the observable behaviour of programs.

[12]The civilised way to do this is to index all our judgements by finite sets of variable names.

$$\vdash X \sim X : \Theta \Rightarrow \texttt{int}\ \text{[A-V]} \qquad \vdash \texttt{n} \sim \texttt{n} : \Theta \Rightarrow \texttt{int}\ \text{[A-N]}$$

$$\vdash \texttt{b} \sim \texttt{b} : \Theta \Rightarrow \texttt{bool}\ \text{[A-B]} \qquad \vdash X \sim E : \Theta \cup \{X = E\} \Rightarrow \texttt{int}\ \text{[A-Red]}$$

$$\vdash \texttt{skip} \sim \texttt{skip} : \Theta \Rightarrow \Theta\ \text{[A-Skp]}$$

$$\frac{\vdash E \sim E' : \Theta \Rightarrow \texttt{int} \quad \vdash F \sim F' : \Theta \Rightarrow \texttt{int}}{\vdash E\ \texttt{iop}\ F \sim E'\ \texttt{iop}\ F' : \Theta \Rightarrow \texttt{int}}\ \text{[A-iop] (+ similar } \textit{bop} \text{ and } \textit{lop}\text{)}$$

$$\frac{\vdash C_1 \sim C_1' : \Theta \Rightarrow \Theta' \quad \vdash C_2 \sim C_2' : \Theta' \Rightarrow \Theta''}{\vdash (C_1 ; C_2) \sim (C_1' ; C_2') : \Theta \Rightarrow \Theta''}\ \text{[A-Seq]}$$

$$\frac{\vdash B \sim B' : \Theta \Rightarrow \texttt{bool} \quad \vdash C \sim C' : \Theta \Rightarrow \Theta}{\vdash (\texttt{while}\ B\ \texttt{do}\ C) \sim (\texttt{while}\ B'\ \texttt{do}\ C') : \Theta \Rightarrow \Theta}\ \text{[A-Whl]}$$

$$\frac{\vdash E \sim E' : \Theta \Rightarrow \texttt{int}}{\vdash X\texttt{:=}E \sim X\texttt{:=}E' : \Theta \Rightarrow (\textit{kill}(\Theta, X) \cup \textit{gen}(X, E) \cup \textit{gen}(X, E'))}\ \text{[A-Ass]}$$

$$\frac{\vdash B \sim B' : \Theta \Rightarrow \texttt{bool} \quad \vdash C_1 \sim C_1' : \Theta \Rightarrow \Theta' \quad \vdash C_2 \sim C_2' : \Theta \Rightarrow \Theta'}{\vdash (\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2) \sim (\texttt{if}\ B'\ \texttt{then}\ C_1'\ \texttt{else}\ C_2') : \Theta \Rightarrow \Theta'}\ \text{[A-If]}$$

$$\frac{\vdash C \sim C' : \Theta \Rightarrow \Theta'}{\vdash C' \sim C : \Theta \Rightarrow \Theta'}\ \text{[A-CSym]}$$

$$\frac{\vdash C \sim C' : \Theta_1 \Rightarrow \Theta_2 \quad \Theta_1' \leq \Theta_1 \quad \Theta_2 \leq \Theta_2'}{\vdash C \sim C' : \Theta_1' \Rightarrow \Theta_2'}\ \text{[A-CSub]}$$

$$\frac{\vdash E_\tau \sim E_\tau' : \Theta \Rightarrow \tau \quad \Theta' \leq \Theta}{\vdash E_\tau \sim E_\tau' : \Theta' \Rightarrow \tau}\ \text{[A-ESub]} \qquad \frac{\vdash F_\tau \sim F_\tau' : \Theta \Rightarrow \tau}{\vdash F_\tau' \sim F_\tau : \Theta \Rightarrow \tau}\ \text{[A-ESym]}$$

$$\frac{\vdash C \sim C' : \Theta \Rightarrow \Theta' \quad \vdash C' \sim C'' : \Theta \Rightarrow \Theta'}{\vdash C \sim C'' : \Theta \Rightarrow \Theta'}\ \text{[A-CTr]}$$

$$\frac{\vdash F_\tau \sim F_\tau' : \Theta \Rightarrow \tau \quad \vdash F_\tau' \sim F_\tau'' : \Theta \Rightarrow \tau}{\vdash F_\tau \sim F_\tau'' : \Theta \Rightarrow \tau}\ \text{[A-ETr]}$$

Figure 11: AERC: Available Expressions and Redundant Computation

Lerner et al. [31, 32] have built a couple of implementations of a domain-specific languages for specifying and justifying rewrites on a simple imperative language which interfaces to a theorem prover for checking the supplied justification. This systems use temporal logic formulae or user-defined propogation rules for specifying the justifications of optimizations over flow graphs.

Kozen and Patron [29] describe an algebraic approach to proving some traditional optimizations correct. There is no mention of relations in their work, and they abstract rather severely from the actual language (there are no assignments, just unspecified atomic programs including one which makes a variable 'undefined'), but the connections between their work and this seem worth further study.

Other work that is closely related to that presented here has been done in the contexts of *credible compilation* [41, 42] and *translation validation* [36, 57]. These both take the view that formal verification of complete optimizing compilers is impractical, but that one might realistically produce a correctness proof relating the input and output of particular compilations. Translation validation tries to do this without modifying the compiler, using an independent tool that tries to infer that the output is a correct translation of the input. Credible compilation envisages an instrumented compiler producing a putative proof that the transformations it performed in each particular case were safe; these proofs can then be examined by a comparatively simple proof-checker. The basic technical ideas used in credible and validated compilation are very close indeed to the ones presented here (developed quite independently). The main difference is that we use the language of types, denotational semantics and PERs instead of that of control-flow graphs, operational semantics and simulation relations. Inspired by Rinard's work, Yang [56] has recently used a version of relational Hoare logic in reasoning about the correctness of the Schorr-Waite graph marking algorithm.

The idea of directly axiomatising a logic of PERs [3] and more general relations was inspired by the work of Abadi et al on a formal logic for parametric polymorphism [2].

We have already mentioned some of the large amount of recent work using PERs (and domain-theoretic projections) to give semantics to analyses for non-interference, slicing, secure information flow, binding time analysis. An elegant general calculus, DCC, for such dependency-based analyses has been defined by Abadi et al. [1]. DCC seems comparable to a higher-order version of our DDCC, though it is not explicitly presented as an equational calculus and is more directly in the style of type systems for secure information flow.

The work of Hughes on type specialization [23] seems to have interesting connections with (a higher-order version of) the work presented here. Hughes has formulated a type-based analysis which essentially uses a form of singleton type, and proved the correctness of an associated transformation system which changes types. Singleton types and their PER semantics have also been studied in some depth by Aspinall [6].

4 An Effect Analysis

In this section we discuss how a very simple effect analysis for a higher-order language may be given an extensional semantics. The work described in this section is particularly preliminary, and a fuller account will be published elsewhere in due course.

Many analyses and logics for imperative programs involve reasoning about whether particular mutable variables (or references or heap cells or regions) may be read or written by a phrase. For example, the equivalence of while-programs

```
C ; if B then C' else C''   =   if B then (C;C') else (C;C'')
```

is valid when `B` does not read any variable which `C` might write. Hoare-style programming logics often have rules with side-conditions on possibly-read and possibly-written variable sets, and reasoning about concurrent processes is dramatically simplified if one can establish that none of them may write a variable which another may read.[13]

Effect systems, first introduced by Gifford and Lucassen [20, 33], are static analyses that compute upper bounds on the possible side-effects of computations. The literature contains many effect systems that analyse which storage cells may be read and which storage cells may be written (as well as many other properties), but no truly satisfactory account of the semantics of this information, or of the the uses to which it may be put. Note that because effect systems *over*estimate the possible side-effects of expressions, the information they capture is of the form that particular variables will definitely *not* be read or will definitely *not* be written. But what does that mean?

Thinking operationally, it may seem entirely obvious what is meant by saying that a variable X will not be read (written) by a command C, viz. no execution trace of C contains a read (resp. write) operation to X. But, as we have already argued such intensional interpretations of program properties are over-restrictive, cannot be interpreted in a standard semantics, do not behave well with respect to program equivalence or contextual reasoning and are hard to maintain during transformations. Thus we seek extensional properties that are more liberal than the intensional ones yet still validate the transformations or reasoning principles we wish to apply.

In the case of not writing a variable, a naive extensional interpretation seems clear: a command C *does not observably write the variable* X if it leaves the value of X unchanged:

$$\forall S, S'.\ C, S \Downarrow S' \implies S'(X) = S(X)$$

though we should note right away that even this definition lacks something as a basis for contextual reasoning: firstly, the quantification over *all* initial states is rather strong (in a particular program context we will often be able to determine static constraints on the possible initial states), and, secondly, the requirement for *equality* of the final values of X is also strong (a context may care only

[13]Though here we restrict attention, in a rather essential manner, to sequential programs.

about, say, the parity of the final value of X). Thus even our obvious definition calls out to be relativized with respect to contexts.

Note that this definition places no constraint on diverging executions or the value of X at intermediate states – operationally, C may read and write X many times, so long as it always restores the original value before terminating. Furthermore, the definition is clearly closed under observational equivalence. If we have no non-termination and just two integer variables, X and Y, and the denotation of C is $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ then our simple-minded definition of what it means for C not to write X can be expressed denotationally as

$$\exists f_2 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}.\ f(X, Y) = (X, f_2(X, Y))$$

The property of *neither reading nor writing* X is also not hard to formalize extensionally:

$$\forall S, S', n.\ C, S \Downarrow S' \iff C, S[X \mapsto n] \Downarrow S'[X \mapsto n]$$

Alternatively

$$\exists f_2 : \mathbb{Z} \to \mathbb{Z}.\ f(X, Y) = (X, f_2(Y))$$

The property of *not observably reading* X is rather more subtle, since X may, or may not, be written. We want to say that the final values of all the other variables are independent of the initial value of X, but the final value of X itself is either a function of the other variables or is the initial value of X:

$$\exists f_1 : \mathbb{Z} \to \mathbb{B}, f_2, f_3 : \mathbb{Z} \to \mathbb{Z}.\ f(X, Y) = (f_1(Y) \implies X \mid f_2(Y), f_3(Y))$$

This is clearly a more complex property than the others! In earlier, operationally-based, work [12] we expressed a global 'does not read' property on integer-valued state using sets of cotermination tests (pairs of contexts) written explicitly in the language, but those definitions were *very* unwieldy and phrased in a way that would not generalize easily to other types. The tricky nature of the does not read property also shows up if one tries to define a family of monads in a synthetic, rather than an analytic fashion: neither reading nor writing corresponds to the identity monad; not writing corresponds to the reader (environment) monad; but there is no nice definition of a 'writer' monad.[14]

The previous section described how such an extensional relational interpretation of static analyses allows one both to express constancy and dependency properties for while-programs, and to reason about the transformations they enable. Here we will show how reading and writing properties for a higher-order language with state can also be captured in a relational framework. The final results we will obtain are similar to (and in fact, for the most part, more restricted than) those of our previous work [12] on proving the correctness of effect-based transformations in the MIL-lite subset of the intermediate language of MLj [13]

[14]Though, chatting to Ian Stark about this, it seems that the general theory of generating computational monads algebraically, from operations and equations [40], might allow one to deduce that such a monad exists in various categories of interest, even if it doesn't have an obvious definition in terms of familar first-order type constructions.

and SML.NET [14]. The difference is that the new semantic interpretation of effects is dramatically slicker and more extensible than in the earlier work.

Our starting point is the following observation, which, though simple, does not seem to have been published before:

Lemma 6.

1. *The above definition of not writing X is equivalent to*

$$\forall R \leq \Delta.\ f : R \times \Delta \rightarrow R \times \Delta$$

2. *The property of neither reading nor writing X is equivalent to*

$$\forall R.\ f : R \times \Delta \rightarrow R \times \Delta$$

3. *The property of not reading X is equivalent to*

$$\forall R \geq \Delta.\ f : R \times \Delta \rightarrow R \times \Delta$$

□

There are many ways we can use the idea above to produce static analysis and transformation systems or program logics. One is to produce a fairly general syntactic system which extends DDCC or RHL with bounded quantification. Another is to present simpler (more first order) effect analyses and transformations and use bounded quantification in giving their semantics and soundness proofs. Of course, we should be able to to both, and justify the effect systems via a syntactic translation into the more generic systems. The details of the systems with explicit bounded quantification in the syntax are fairly complex, so we'll start by just doing the second.

4.1 The World's Second-Stupidest Effect System

4.1.1 Base Language

We consider a monadically-typed, normalizing, call-by-value lambda calculus with a single global integer reference. This extremely simple setting allows us to explore the key idea without getting bogged down in too much auxiliary detail.

Value types A, computation types TA and contexts Γ:

$$\begin{array}{rcl} A, B & := & \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid A \times B \mid A \rightarrow TB \\ \Gamma & := & x_1 : A_1, \ldots, x_n : A_n \end{array}$$

Value judgements $\Gamma \vdash V : A$ and computation judgements $\Gamma \vdash M : TA$ are shown in Figure 12. Denotational semantics just uses sets as no recursion.

$$\frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 + V_2 : \mathtt{int}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{int} \quad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 > V_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i\, V : A_i}$$

$$\frac{\Gamma, x : A \vdash M : TB}{\Gamma \vdash \lambda x : A.M : A \to TB} \qquad \frac{\Gamma \vdash V_1 : A \to TB \quad \Gamma \vdash V_2 : A}{\Gamma \vdash V_1\, V_2 : TB}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{val}\, V : TA} \qquad \frac{\Gamma \vdash M : TA \quad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N : TB}$$

$$\frac{\Gamma \vdash V : \mathtt{bool} \quad \Gamma \vdash M : TA \quad \Gamma \vdash N : TA}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : TA}$$

$$\frac{}{\Gamma \vdash \mathtt{read} : T\mathtt{int}} \qquad \frac{\Gamma \vdash V : \mathtt{int}}{\Gamma \vdash \mathtt{write}(V) : T\mathtt{unit}}$$

Figure 12: Simple computation type system

Write S for $\mathbb{Z}$.

$$\begin{array}{rcl} [\![\mathtt{unit}]\!] & = & 1 \\ [\![\mathtt{int}]\!] & = & \mathbb{Z} \\ [\![\mathtt{bool}]\!] & = & \mathbb{B} \\ [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\ [\![A \to TB]\!] & = & [\![A]\!] \to [\![TB]\!] \\ [\![TA]\!] & = & S \to S \times [\![A]\!] \end{array}$$

Semantics of contexts:

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

Semantics of terms in context:

$$\begin{array}{rcl} [\![\Gamma \vdash V : A]\!] & : & [\![\Gamma]\!] \to [\![A]\!] \\ [\![\Gamma \vdash M : TA]\!] & : & [\![\Gamma]\!] \to [\![TA]\!] \end{array}$$

Defined inductively in the usual way. Types on binders make derivations unique. Adequate for the obvious operational semantics and notion of equivalence.

$$\frac{}{X \leq X} \qquad \frac{X \leq Y \quad Y \leq Z}{X \leq Z} \qquad \frac{X \leq X' \quad Y \leq Y'}{X \times Y \leq X' \times Y'}$$

$$\frac{X' \leq X \quad T_\varepsilon Y \leq T_{\varepsilon'} Y'}{(X \to T_\varepsilon Y) \leq (X' \to T_{\varepsilon'} Y')} \qquad \frac{\varepsilon \subseteq \varepsilon' \quad X \leq X'}{T_\varepsilon X \leq T_{\varepsilon'} X'}$$

Figure 13: Subtyping extended types

4.1.2 Effect system

Extended value types X, computation types $T_\varepsilon X$ and contexts Θ:

$$\begin{array}{rcl} X, Y & := & \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid X \times Y \mid X \to T_\varepsilon Y \\ \varepsilon & \subseteq & \{\texttt{r}, \texttt{w}\} \\ \Theta & := & x_1 : X_1, \ldots, x_n : X_n \end{array}$$

Subtyping on extended types shown in Figure 13.

There's an erasure map $U(\cdot)$, taking extended types to simple types by forgetting the effect annotations:

$$\begin{array}{rcl} U(\texttt{int}) & = & \texttt{int} \\ U(\texttt{bool}) & = & \texttt{bool} \\ U(\texttt{unit}) & = & \texttt{unit} \\ U(X \times Y) & = & U(X) \times U(Y) \\ U(X \to T_\varepsilon Y) & = & U(X) \to U(T_\varepsilon Y) \\ U(T_\varepsilon X) & = & T(U(Y)) \end{array}$$

which we extend pointwise to contexts:

$$U(x_1 : X_1, \ldots, x_n : X_n) = x_1 : U(X_1), \ldots, x_n : U(X_n)$$

The following is obvious:

Lemma 7. *If $X \leq Y$ then $U(X) = U(Y)$, and similarly for computations.* □

The extended type assignment system is shown in Figure 14. Note that the terms are the same (we still only have simple types on λ-bound variables). The following is a trivial induction:

Lemma 8. *If $\Theta \vdash V : X$ then $U(\Theta) \vdash V : U(X)$, and similarly for computations.* □

$$\frac{}{\Theta \vdash n : \texttt{int}} \qquad \frac{}{\Theta \vdash b : \texttt{bool}} \qquad \frac{}{\Theta \vdash () : \texttt{unit}} \qquad \frac{}{\Theta, x : X \vdash x : X}$$

$$\frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 + V_2 : \texttt{int}} \qquad \frac{\Theta \vdash V_1 : \texttt{int} \quad \Theta \vdash V_2 : \texttt{int}}{\Theta \vdash V_1 > V_2 : \texttt{bool}}$$

$$\frac{\Theta \vdash V_1 : X \quad \Theta \vdash V_2 : Y}{\Theta \vdash (V_1, V_2) : X \times Y} \qquad \frac{\Theta \vdash V : X_1 \times X_2}{\Theta \vdash \pi_i\, V : X_i}$$

$$\frac{\Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon Y} \qquad \frac{\Theta \vdash V_1 : X \to T_\varepsilon Y \quad \Theta \vdash V_2 : X}{\Theta \vdash V_1\, V_2 : T_\varepsilon Y}$$

$$\frac{\Theta \vdash V : X}{\Theta \vdash \texttt{val}\, V : T_\emptyset X} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M\, \texttt{in}\, N : T_{\varepsilon \cup \varepsilon'} Y}$$

$$\frac{\Theta \vdash V : \texttt{bool} \quad \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon X}{\Theta \vdash \texttt{if}\ V\ \texttt{then}\ M\ \texttt{else}\ N : T_\varepsilon X}$$

$$\frac{}{\Theta \vdash \texttt{read} : T_{\{\texttt{r}\}}\texttt{int}} \qquad \frac{\Theta \vdash V : \texttt{int}}{\Theta \vdash \texttt{write}(V) : T_{\{\texttt{w}\}}\texttt{unit}}$$

$$\frac{\Theta \vdash V : X \quad X \leq X'}{\Theta \vdash V : X'} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \leq T_{\varepsilon'} X'}{\Theta \vdash M : T_{\varepsilon'} X'}$$

Figure 14: Extended type system

4.1.3 Semantics of Effects

Now we define the semantics of extended types as binary relations on the semantics of their erasures.

$$\begin{array}{rcl}
[\![X]\!] & \subseteq & [\![U(X)]\!] \times [\![U(X)]\!] \\
[\![\mathtt{int}]\!] & = & \Delta_{\mathbb{Z}} \\
[\![\mathtt{bool}]\!] & = & \Delta_{\mathbb{B}} \\
[\![\mathtt{unit}]\!] & = & \Delta_1 \\
[\![X \times Y]\!] & = & [\![X]\!] \times [\![Y]\!] \\
[\![X \to T_\varepsilon Y]\!] & = & [\![X]\!] \to [\![T_\varepsilon Y]\!] \\
[\![T_\varepsilon X]\!] & = & \bigcap_{R \in \mathcal{R}_\varepsilon} R \to R \times [\![X]\!]
\end{array}$$

where

$$\begin{array}{rcl}
\mathcal{R}_{\{\}} & = & \mathbb{P}(S \times S) \\
\mathcal{R}_{\{\mathtt{r}\}} & = & \{R \mid R \subseteq \Delta_S\} \\
\mathcal{R}_{\{\mathtt{w}\}} & = & \{R \mid R \supseteq \Delta_S\} \\
\mathcal{R}_{\{\mathtt{r},\mathtt{w}\}} & = & \{\Delta_S\}
\end{array}$$

For each ε there is a set $\mathcal{R}_\varepsilon$ of relations on the state that computations of type $T_\varepsilon X$ have to preserve; the more possible effects occur in ε, the fewer relations are preserved.

We also extend the relational interpretation of extended types to extended contexts in the natural way:

$$\begin{array}{rcl}
[\![\Theta]\!] & \subseteq & [\![U(\Theta)]\!] \times [\![U(\Theta)]\!] \\
[\![x_1 : X_1, \ldots, x_n : X_n]\!] & = & \{(\rho, \rho') \mid \forall 1 \leq i \leq n.\ (\pi_i(\rho), \pi_i(\rho')) \in [\![X_1]\!]\}
\end{array}$$

Lemma 9. *For any Θ, X and ε, all of $[\![\Theta]\!]$, $[\![X]\!]$ and $[\![T_\varepsilon X]\!]$ are partial equivalence relations.* □

The following establishes semantic soundness for our subtyping relation:

Lemma 10. *If $X \leq Y$ then $[\![X]\!] \subseteq [\![Y]\!]$, and similarly for computation types.* □

And we can then show a 'fundamental theorem' establishing the soundness of the effect analysis itself:

Theorem 8.

1. *If $\Theta \vdash V : X$, $(\rho, \rho') \in [\![\Theta]\!]$ then*

$$([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V : U(X)]\!]\,\rho') \in [\![X]\!]$$

2. *If* $\Theta \vdash M : T_\varepsilon X$, $(\rho, \rho') \in [\![\Theta]\!]$ *then*

$$([\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho,\ [\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho') \in [\![T_\varepsilon X]\!]$$

□

The soundness proofs above are very straightforward and familiar-looking, which is one of the benefits of our relational formulation. Because we have used standard technology (logical relations, PERs), the pattern of what we have to prove is obvious and the definitions are all set up so that the proofs go through smoothly. Had we defined the semantics of effects in some more special-purpose way (e.g. trying to work directly with the property of being uniformly either constant or the identity), it could have been rather less obvious how to make everything extend smoothly to higher-order and how to deal with combining effects in the let-rule.

4.1.4 Using Effect Information

This section discusses how effect information may be used to justify program transformations, presented as typed equations in context.

Before looking at effect-dependent equivalences, we note that the semantics validates all the usual equations of the computational metalanguage, including congruence laws and β and η laws for products, function spaces, booleans and the computation type constructor. The correctness of the basic congruence laws subsumes Theorem 8.

More interesting equivalences are predicated on the effect information. We show some of these in Figure 15.

Theorem 9. *All of the equations shown in Figure 15 are soundly modelled in the semantics, in the sense that*

- *If* $\Theta \vdash V = V' : X$ *then for all* $(\rho, \rho') \in [\![\Theta]\!]$

$$([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V' : U(X)]\!]\,\rho') \in [\![X]\!]$$

- *If* $\Theta \vdash M = M' : T_\varepsilon X$ *then for all* $(\rho, \rho') \in [\![\Theta]\!]$

$$([\![U(\Theta) \vdash M : T(U(X))]\!]\,\rho,\ [\![U(\Theta) \vdash M' : T(U(X))]\!]\,\rho') \in [\![T_\varepsilon X]\!]$$

□

4.2 Discussion

We have shown how an extensional interpretation of read and write effects may be given using bounded quantification over relations, and how that semantics may be used to enable program transformations.

It is interesting to note that, whilst quantifier-free RHL can express many interesting analyses, it does not admit a simple compositional translation of

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N = N : T_{\varepsilon'} Y}\; x \notin \Theta, \varepsilon \subseteq \{\mathtt{r}\}$$

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X, y : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{rl} & \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, \mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, N \\ = & \mathtt{let}\, x \Leftarrow M\, \mathtt{in}\, N[x/y] \end{array} : T_{\varepsilon \cup \varepsilon'} Y}\; \varepsilon \subseteq \{\mathtt{r}\} \; \mathit{or} \; \varepsilon \subseteq \{\mathtt{w}\}$$

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X_1 \quad \Theta \vdash M_2 : T_{\varepsilon_2} X_2 \quad \Theta, x_1 : X_1, x_2 : X_2 \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{rl} & \mathtt{let}\, x_1 \Leftarrow M_1\, \mathtt{in}\, \mathtt{let}\, x_2 \Leftarrow M_2\, \mathtt{in}\, N \\ = & \mathtt{let}\, x_2 \Leftarrow M_2\, \mathtt{in}\, \mathtt{let}\, x_1 \Leftarrow M_1\, \mathtt{in}\, N \end{array} : T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon'} Y}\; \begin{array}{l} \varepsilon_1, \varepsilon_2 \subseteq \{\mathtt{r}\} \\ \mathit{or}\; \varepsilon_1 = \{\} \end{array}$$

$$\frac{\Theta \vdash M : T_{\{\}} Z \quad \Theta, x : X, y : Z \vdash N : T_\varepsilon Y}{\Theta \vdash \begin{array}{rl} & \mathtt{val}\,(\lambda x : U(X).\mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, N) \\ = & \mathtt{let}\, y \Leftarrow M\, \mathtt{in}\, \mathtt{val}\,(\lambda x : U(X).N) \end{array} : T_{\{\}}(X \to T_\varepsilon Y)}$$

Figure 15: Effect-dependent equivalences

perhaps the simplest static analysis there is for while-programs, viz. the obvious inductive definition of possibly-read and possibly-written sets for each command. The results here show that adding relation variables and quantification to RHL would certainly solve that problem, but the details of the best way to define such an extension deserve some further study.

The tricky case for our semantics was the 'does not read' property, which we interpreted using relational quantification with a lower bound. Whilst quantification with upper bounds, which we used to interpret 'does not write', is fairly common in the literature on type systems, lower bounds have received rather less attention.

It is worth remarking that an apparently appealing candidate for an extensional characterization of not reading is *idempotency*. For simple commands $C : S \to S$, it is easy to see that a command that does not read the store, i.e. that is either constant or the identity, satisfies $C; C = C$. Unfortunately this does not work as a semantics of not reading, as idempotency is not closed under sequential composition (in Führmann's [19] terminology, idempotency is not an *effectoid*).

Of course, a terminating language with a single piece of integer-valued global state is not particularly exciting or realistic, and there are many ways in which this effect system and language could be extended, including higher types in the store, dynamic allocation, regions, effect polymorphism and combining state with other notions of computation. However, there is good reason to believe that the basic techniques used here will extend smoothly to such settings.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Conference Record of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 147–160. ACM Press, January 1999.

[2] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121, 1993.

[3] M. Abadi and G. D. Plotkin. A PER model of polymorphism and recursive types. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 355–365. IEEE Computer Society Press, June 1990.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[5] T. Amtoft. Minimal thunkification. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, September 1993.

[6] D. Aspinall. Subtyping with singleton types. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th International Workshop (CSL'94)*, number 933 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

[7] C. A. Baker-Finch. Relevant logic and strictness analysis. In *Workshop on Static Analysis, LaBRI, Bordeaux.* Bigre, 1992.

[8] N. Benton. A unified approach to strictness analysis and optimizing transformations. Technical Report 388, Computer Laboratory, University of Cambridge, February 1996.

[9] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, January 2004. Revised version available from `http://research.microsoft.com/~nick/publications.htm`.

[10] N. Benton. Semantics of program analyses and transformations. Lecture Notes for the PAT Summer School, Copenhagen, June 2005.

[11] N. Benton. Simple relational correctness proofs for static analyses and program transformations. Technical Report MSR-TR-2005-26, Microsoft Research, February 2005.

[12] N. Benton and A. Kennedy. Monads, effects and transformations. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS), Paris*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1999.

[13] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming (ICFP)*, September 1998.

[14] N. Benton, A. Kennedy, and C. Russo. Adventures in interoperability: The SML.NET experience. In *Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, August 2004.

[15] P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, Computer Laboratory, University of Cambridge, December 1992.

[16] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proceedings of the Second International Symposium on Logical Foundations of Computer Science, Tver, Russia*, volume 620 of *Lecture Notes in Computer Science*, pages 33–44. Springer-Verlag, July 1992.

[17] F. Damiani. Useless-code Detection and Elimination for PCF with Algebraic Datatypes. In *4th International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 1581 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1999.

[18] F. Damiani and P. Giannini. Automatic useless-code detection and elimination for hot functional programs. *Journal of Functional Programming*, pages 509–559, 2000.

[19] C. Führmann. Varieties of effects. In *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2002.

[20] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, Massachusetts, August 1986.

[21] M S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[23] J. Hughes. Type specialization for the lambda calculus. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, 1996.

[24] S. Hunt and D. Sands. Binding time analysis: A new PERspective. In *Proceedings ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, June 1991.

[25] T. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, University of London, November 1992.

[26] T. P. Jensen. Strictness analysis in logical form. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, 1991.

[27] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 194–206. ACM Press, 1973.

[28] N. Kobayashi. Type-based useless variable elimination. In *Proceeedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2000.

[29] D. Kozen and M. Patron. Certification of compiler optimizations using Kleene algebra with tests. In *Proceedings of the 1st International Conference in Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

[30] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Proceedings of the 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, Portland*, January 2002.

[31] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[32] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Conference Record of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2005.

[33] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1988.

[34] M. Mizuno and D.A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4:727–754, 1992.

[35] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281. Springer-Verlag, April 1980.

[36] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 83–95, 2000.

[37] H. R. Nielson and F. Nielson. Context information for lazy code generation. In *LISP and Functional Programming*, June 1990.

[38] P. Orbaek. Can you trust your data? In *Proceedings of TAPSOFT/FASE*, volume 915 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[39] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[40] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures, Proceedings of FOSSACS '02*, volume 2303 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[41] M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusets Institute of Technology, March 1999.

[42] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, July 1999.

[43] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[44] Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Frank Report 19, Institut für Informatik, J.W. Goethe-Universität Frankfurt am Main, October 2004.

[45] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, January 1998.

[46] P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 48–86, January 1997. Original version appeared in Proceedings 21st ACM Symposium on Principles of Programming Languages, 1994.

[47] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, December 1996.

[48] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.

[49] M. Wand and W. D. Clinger. Set constraints for destructive array update optimization. *Journal of Functional Programming*, 11(3):319–346, May 2001. Preliminary version appeared in International Conference on Computer Languages, 1998.

[50] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proceedings 26th ACM Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[51] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.

[52] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[53] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[54] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[55] D. A. Wright. *Reduction Types and Intensionality in the Lambda-calculus.* PhD thesis, University of Tasmania, 1992.

[56] H. Yang. Verification of the Schorr-Waite graph marking algorithm by refinement. Slides from talk at Dagstuhl Seminar 03101, March 2003.

[57] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation for optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.